

PRIMO CAPITOLO

Una delle caratteristiche fondamentali da acquisire è l'approccio sistematico alla gestione della complessità. La tecnica fondamentale è l'**astrazione**, che consiste nel nascondere dettagli quando questi non sono importanti. I progettisti usano le tre -Y:

- La **gerarchia** implica dividere un sistema in moduli e successivamente suddividere ulteriormente ognuno di questi in moduli, fino a raggiungere un livello in cui ognuno di essi sia comprensibile (semplificare gerarchicamente il sistema/problema in pezzi più semplici).
- La **modularità** implica che i moduli abbiano funzioni e interfacce ben definite, così da connettersi tra di loro in maniera semplice e senza effetti collaterali inaspettati (parti del sistema possono essere modificate con l'unica condizione che non alterino la loro funzione, compromettendo l'organicità dell'intero sistema).
- La **regolarità** cerca l'uniformità tra i moduli (in questo modo pezzi di altre manifatture con la stessa funzione possono essere sostituiti in caso di necessità senza alterare il funzionamento del sistema).

I sistemi digitali rappresentano informazioni con **variabili dal valore discreto**, cioè con un numero finito di valori possibili. La quantità di informazione D associata a una variabile a valori discreti con N stati distinti è misurata in termini di bit (binary digit) come:

$$D = \log_2 N \text{ bit}$$

Ad esempio una singola variabile esadecimale rappresenta 4 bit di informazione ($\log 16=4$).

Nel sistema decimale a cui siamo abituati ci sono dieci cifre e nel momento in cui le andiamo a combinare, ogni posizione di una cifra ha 10 volte il peso della posizione alla sua destra. I bit rappresentano uno dei due valori 0 o 1, e vengono uniti l'uno con l'altro per formare numeri binari; ogni posizione di un bit di un numero binario ha il doppio del peso della posizione precedente. Un numero binario a N bit può rappresentare 2^N possibili valori.

Se il risultato di una somma tra due numeri binari è troppo grande per essere espresso con le cifre a disposizione, si dice che la somma da luogo a un **overflow**.

Rappresentazione modulo e segno

I numeri in **modulo e segno** utilizzano il bit più significativo per esprimere il segno e i rimanenti bit come indicatori del modulo. Per quanto possa essere intuitiva questa rappresentazione, essa presenta due possibili rappresentazioni dello zero, il che produce risultati erronei nelle operazioni che li attraversano.

Rappresentazione in complemento a due

Nella rappresentazione in **complemento a due** il massimo numero positivo ha il bit più significativo a 0 e tutti gli altri a 1, e il minimo numero negativo ha il bit più significativo a 1 e tutti gli altri a 0. Se immaginiamo una linea su cui posizionare i numeri, questi si troverebbero esattamente nel lato opposto in cui li collociamo solitamente, con i positivi a sinistra e i negativi a destra.

Si noti che questo tipo di rappresentazione permette ancora di avere un bit dedicato al segno che renda più intuitiva la lettura del numero, ma questo viene comunque interpretato in maniera differente: per i numeri positivi basta la normale conversione in decimale, ma per quelli negativi, per scoprire il modulo, bisogna invertire il segno con il calcolo del complemento a due, che consiste nell'invertire tutti i bit e poi aggiungere uno.

Sistema	Range
Senza segno	$[0, 2^N - 1]$
Modulo e segno	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Complemento a due	$[-2^{N-1}, 2^{N-1} - 1]$

Quando si esegue una somma che non comporta overflow ma che ha un riporto nell' N -esimo bit, questo deve essere scartato. Per la sottrazione basta effettuare il complemento a due del secondo numero e poi procedere con la somma. Si noti inoltre che lo 0 ha un'unica rappresentazione, quindi il problema che poteva insorgere con modulo e segno è eliminato.

Rappresentazione in virgola mobile

Per rappresentare i numeri razionali possiamo utilizzare la notazione in virgola fissa o in virgola mobile. La prima è uguale ai numeri decimali e si basa su una virgola fissa implicita tra i bit della parte intera e quelli della parte frazionaria; i numeri con segno possono essere rappresentati sia in complemento a due che in segno e modulo. Con questa rappresentazione però non c'è modo di individuare la virgola se non attraverso un accordo preventivo.

La rappresentazione in virgola mobile è equivalente alla rappresentazione scientifica, e per questo a ogni numero è assegnato un **segno**, una **mantissa**, una **base** e un **esponente**, a cui vengono assegnati rispettivamente 1, 8 e 23 bit, per un totale di 32; la base è ovviamente 2.

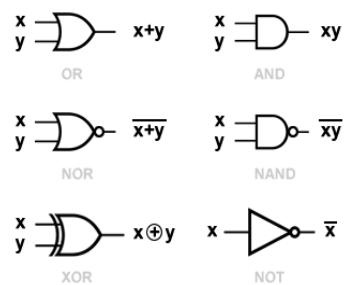
Piccole attenzioni che devono essere fatte è che nei numeri in virgola mobile il primo bit della mantissa è sempre un 1, quindi non ha bisogno di essere memorizzato, e l'esponente deve essere in grado di rappresentare sia numeri positivi che negativi, quindi viene utilizzata la codifica a eccesso, costituita dall'esponente principale più la costante 127.

Analizziamo i sistemi digitali che effettuano operazioni su queste variabili binarie. Le **porte logiche** sono semplici circuiti digitali che utilizzano uno o più ingressi binari per produrre un'uscita binaria. La relazione tra ingressi e uscita può essere descritta con una tabella di verità o con un'espressione booleana.

L'altra porta logica a un solo ingresso viene chiamata buffer e riproduce semplicemente il valore di ingresso in uscita. Dal punto di vista logico un buffer non è diverso da un semplice filo, ma ha invece caratteristiche utili, come trasmettere velocemente il valore della sua uscita a molte porte logiche diverse. Questo è un esempio di come l'astrazione digitale nasconde l'utilità reale del buffer.

Il circoletto di negazione viene chiamato bolla. Il **bubble pushing** è un metodo grafico per semplificare i circuiti: "spingere" una bolla attraverso una porta fa sì che la bolla fuoriesca dall'altro lato e trasformi il corpo della porta da AND a OR, o viceversa. Questo metodo applica le leggi di De Morgan e la legge della doppia negazione. È importante ricordare che una bolla dall'uscita viene trasferita a tutti gli ingressi della porta di appartenenza.

Un sistema digitale utilizza variabili a valori discreti, ma sono rappresentate da quantità fisiche continue come la tensione elettrica o il livello di un fluido in un cilindro. Per questo motivo ogni sistema reale deve tollerare una certa quantità di rumore. Preso ad esempio un sistema con un generatore e un ricevitore, il generatore ha degli intervalli che definiscono i livelli logici dei valori basso e alto, e il ricevitore ha un **margine di rumore** che è la quantità di rumore che può essere aggiunta all'uscita nella peggiore delle ipotesi per far sì che il segnale riesca a essere interpretato come un ingresso valido.



SECONDO CAPITOLO

Un **circuito** è una rete elettrica che elabora variabili a valori discreti, e contiene:

- uno o più ingressi a valori discreti;
- una o più uscite a valori discreti;
- una specifica funzionale che descrive la relazione tra ingressi e uscite;
- una specifica di temporizzazione che descrive il ritardo tra il cambio degli ingressi e la risposta delle uscite.

Le reti sono composte da **elementi**, che sono a loro volta reti con ingressi, uscite e specifiche proprie, e da **nodi**, ovvero contatti elettrici la cui tensione trasmette una variabile a valore discreto, e possono essere di tre tipi: ingressi, uscite e nodi interni. Gli ingressi ricevono valori dal mondo esterno, le uscite emettono valori all'esterno; i contatti che non sono né ingressi né uscite vengono chiamati nodi interni.

Le reti digitali vengono divise in due categorie: reti **combinatorie** e reti **sequenziali**. Le uscite di una rete combinatoria dipendono esclusivamente dai valori presenti in quel momento agli ingressi; le uscite di una rete sequenziale invece dipendono sia dai valori presenti agli ingressi, sia dai valori precedenti. L'indicazione LC indica che la rete è realizzata utilizzando unicamente logica combinatoria; spesso esistono più realizzazioni diverse per una singola funzione. Una singola linea con un trattino che la attraversa è un numero scritto al suo fianco per indicare un **bus**, cioè un insieme di segnali multipli.

Le regole della composizione combinatoria insegnano come combinare elementi di reti combinatorie di dimensioni limitate per creare una rete combinatoria più grande, e sono:

- ogni elemento circuitale è di per sé combinatorio;
- la rete non contiene percorsi ciclici: ogni percorso che la attraversa passa attraverso ogni nodo al massimo una volta;
- ogni nodo che non è un input connette esattamente un output di un elemento.

George Boole ha sviluppato una logica che opera su variabili binaria, nota come logica booleana. Ognuna delle variabili può assumere solo uno dei valori *vero* o *falso*, che i calcolatori rappresentano mediante la tensione elettrica. Il complemento di una variabile è il suo negato; la variabile o il suo complemento sono denominati **letterali**. L'*AND* è definito come prodotto logico o **implicante**, e un **mintermine** è il prodotto di tutti gli ingressi di una funzione; l'*OR* è definito somma logica o **implicato**, e un **maxtermine** è la somma di tutti gli ingressi di una funzione.

Ogni riga di una tabella di verità è associata a mintermine che è VERO per quella riga; è possibile scrivere un'espressione booleana tramite la somma di tutti i mintermini in corrispondenza dei quali l'unica vale VERO (*forma SOP*). Ogni riga di una tabella è anche associata a un maxtermine che è FALSO per quella riga; è possibile scrivere un'espressione tramite il prodotto dei maxtermini in corrispondenza dei quali l'uscita è FALSO (*forma POS*).

La forma SOP viene chiamata **logica a due livelli** perché consiste di letterali connessi a un primo livello di porte AND che, a loro volta, sono connesse a un secondo livello di porte OR.

È possibile utilizzare l'algebra booleana per semplificare le espressioni booleane: i postulati e i teoremi di quest'algebra obbediscono al principio di dualità, ovvero se i simboli 0 e 1 e gli operandi AND e OR sono scambiati tra loro, l'affermazione rimane corretta.

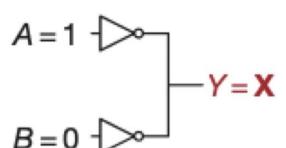
Un implicante è detto **implicante primo** se non può essere combinato con nessun altro elemento all'interno dell'espressione per formare un nuovo implicante con un numero minore di letterali. Un'**espressione minima** se tutti i suoi implicanti sono primi, ovvero se ne utilizza il minor numero possibile.

La semplificazione è importante perché riduce il numero di porte necessarie per eseguire una funzione, rendendola più piccola, meno costosa e probabilmente anche più veloce.

I **circuiti a priorità** vengono utilizzati per assegnare una risorsa condivisa secondo un ordine di priorità di chi ne fa richiesta. Particolarmente utili in questi circuiti sono i valori **don't care** o **indifferenze**, che indicano degli ingressi i cui valori vengono ignorati per determinare l'uscita della funzione; si possono trovare anche in output.

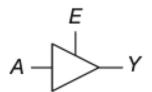
Il simbolo X può indicare anche che il nodo della rete ha un **valore illegale**, che in generale si presenta quando vengono applicati contemporaneamente i valori 0 e 1; questa situazione si chiama **conflitto** ed è un errore che deve essere evitato, perché spesso il risultato è che la tensione elettrica ricada nella zona proibita, e ciò porta la rete a surriscaldarsi e probabilmente a danneggiarsi.

Il simbolo Z indica che un nodo non è portato né a un valore alto, né a uno basso; questo nodo è detto **fluttuante**, e può avere una qualsiasi tensione compresa tra 0 e 1. A differenza di un nodo dal valore illegale, un valore fluttuante non è sempre sintomo di errore, finché è presente un altro elemento della rete che riporta il



nodo a un valore logico valido quando tale valore è rilevante per il funzionamento della rete. I valori fluttuanti vengono utilizzati per disconnettere una parte di circuito dal resto.

Un circuito che realizza tale funzione è il **buffer tristate**. Esso è composto di un ingresso, un'uscita e un'**abilitazione**; quando l'abilitazione è alto (1) il buffer lavora come un normale buffer e trasferisce il valore dell'ingresso all'uscita; quanto è basso (0) l'uscita può avere valore fluttuante.

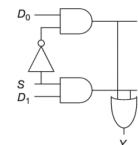
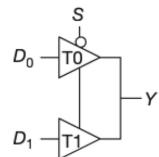
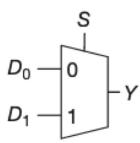


Le Mappe di Karnaugh sono uno strumento molto efficace per semplificare espressioni con al massimo quattro variabili; sfruttano il principio della combinazione, per cui $PA+P(A)^*=P$, e il codice Gray 00,01,11,10 in cui gli elementi adiacenti differiscono per una sola variabile. I cerchi più larghi possibili rappresentano gli implicanti primi.

Multiplexer

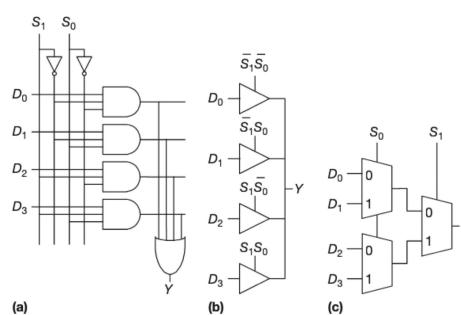
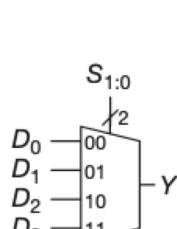
I multiplexer sono in grado di scegliere un'uscita a partire da un certo numero di ingressi possibili basandosi sul valore di un segnale di selezione.

Un multiplexer 2:1 ha due ingressi di dato, un ingresso di selezione e un'uscita; è può essere costruito a partire dalla logica a due livelli (quella della forma SOP), o con buffer tristate.



Un multiplexer 4:1 possiede invece quattro ingressi di dato, e sono necessari due ingressi di selezione per scegliere tra i quattro dati di ingresso. Può essere costruito utilizzando la logica a somma di prodotti, i tristate, o alcuni multiplexer 2:1.

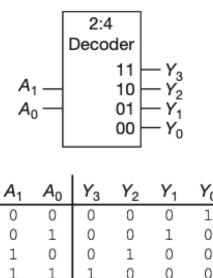
In generale un multiplexer $N:1$ necessita $\log N$ ingressi di selezione. Con un po' di astuzia è possibile dimezzare la taglia del multiplexer, utilizzando solo un multiplexer a $2^{(N-1)}$ ingressi per eseguire una qualsiasi funzione



logica a N ingressi. La strategia di base è fornire uno dei letterali di ingresso della funzione agli ingressi di dato della funzione.

Decoder

Un decoder ha N ingressi e 2^N uscite e attiva una delle uscite a seconda della combinazione di valori di ingresso; le uscite sono dette one hot proprio perché solo un'uscita è "calda" in ogni momento. La logica a decoder consiste nell'utilizzare un decoder con una porta OR per esprimere una funzione in forma SOP; in generale una funzione a N ingressi con un numero M di 1 nella tabella della verità, può essere costruita con un decoder $N:2^N$ e una porta OR a M ingressi connessa a tutti i mintermini associati a un 1 in uscita nella tabella.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

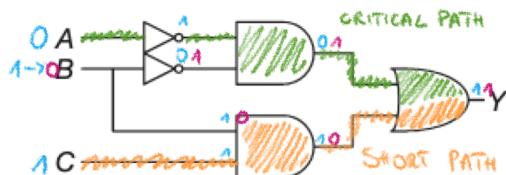
Uno dei problemi più importanti legato alle reti è la temporizzazione: in altre parole come fare in modo che la rete funzioni velocemente. Un'uscita impiega un certo tempo ad adattarsi a un cambiamento avvenuto in un ingresso. Il passaggio da BASSO a ALTO viene chiamato **fronte di salita**, mentre il passaggio inverso viene detto **fronte di discesa**. Il ritardo viene misurato a partire del punto al 50% del segnale di ingresso fino al punto al 50% del

segnale d'uscita. La logica combinatoria è caratterizzata da un **ritardo di propagazione**, che è il tempo massimo che trascorre dal momento in cui avviene un cambiamento nell'ingresso al momento in cui l'uscita raggiunge il suo valore finale; e dal **ritardo di contaminazione**, che è il tempo minimo che trascorre dal momento in cui cambia un ingresso al momento in cui una qualsiasi uscita comincia il processo di adattamento del suo valore. I ritardi sono determinati anche dal percorso che un segnale segue tra l'ingresso e l'uscita.

Il **percorso critico** è quello che passa per il maggior numero di porte, e viene definito critico perché limita la velocità a cui la rete opera. Il **percorso minimo** è il percorso più breve possibile (e quindi anche il più veloce).

Il ritardo di propagazione di una rete combinatoria è uguale alla somma dei singoli ritardi di propagazione di ogni elemento del percorso critico; il ritardo di contaminazione è la somma dei ritardi di contaminazione attraverso ogni elemento del percorso minimo.

È anche possibile che un singolo cambiamento in ingresso causi molteplici cambiamenti in uscita, denominati **alee**; nonostante queste di solito non creino problemi, è importante rendersi conto della loro esistenza e riconoscerle quando si guarda il diagramma temporale.



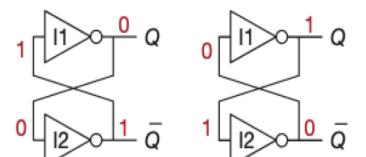
Theoretically, the transition of B from 1 to 0 should not change the output since at the logical level it is always one; at the physical level, however, the critical path AND gate updates after the minimum path, so for some instants the first is still 0 and the second updates to 0 to provide the final output 0.

The ales do not represent a problem if one expects that the propagation delay is exhausted before observing the output value, because this occurs either before or after returning to the correct value.

TERZO CAPITOLO

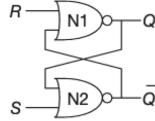
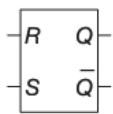
The outputs of **sequential networks** depend on the values present at the moment of the inputs as well as on previous values; for this reason it is said that it has memory. Sequential logic can have the ability to summarize previous inputs in quantities much smaller than information called **states** of the system, otherwise a set of bits, called **variables of state**, which contains all the information necessary to explain the future behavior of the network.

The fundamental building block of memory is a **bistable** element, that is, an element with two stable states; one **state** is **stable** if the initial premise is confirmed, that is, if the value of a variable is assumed, this remains consistent. When a sequential network is turned on, its initial state is unknown and generally unpredictable, and can be different at each new power-on of the network.



Latch SR

The SR latch represents one of the simplest sequential networks; it is composed of two NOR gates connected in cross-coupling, and is similar to the inverters connected in cross-coupling, but its state can be controlled by the inputs **S** and **R** which set (set) or reset (reset) the output **Q**. Setting a bit means assuming its value to be TRUE; resetting it means assuming its value to be FALSE. When neither of the two inputs is activated, **Q** remembers its previous value.

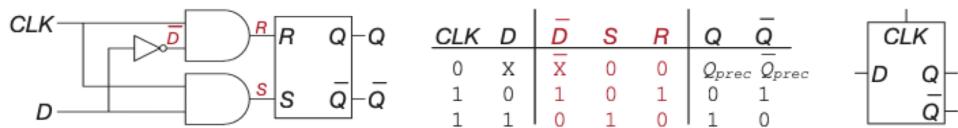


Caso	S	R	Q	\bar{Q}
IV	0	0	Q_{prec}	Q_{prec}
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Latch D

Il latch SR è scomodo in quanto si comporta in maniera imprevedibile quando entrambi gli ingressi sono attivati simultaneamente; inoltre gli ingressi combinano gli aspetti del "come" e del "quando" avviene il cambiamenti di stato, ma la progettazione delle reti diventa più facile quando questi aspetti sono separati.

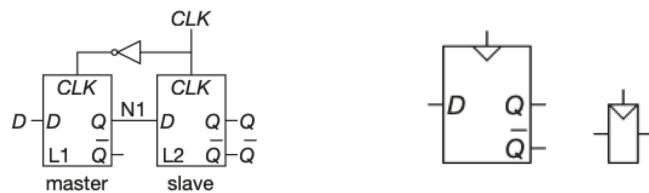
Il latch RD ha due ingressi: un ingresso dati, che controlla il prossimo stato, e un ingresso **clock**, che controlla invece il momento del cambio di stato. Quando $CLK=1$, il latch è detto **trasparente**, e i dati scorrono da D verso Q , come se il latch fosse il buffer. Quando invece $CLK=0$, il latch è **opaco** e viene bloccato il passaggio dei dati verso Q , che mantiene lo stato precedente.



Flip-Flop D

Un flip-flop D può essere costruito a partire da due latch D controllati da due segnali di clock complementari. Il primo latch $L1$ viene detto **master**, mentre il secondo latch $L2$ viene detto **slave**. Quando $CLK=0$, il latch master è trasparente, mentre il latch slave è opaco, quindi il valore dato di ingresso viene portato al nodo tra i latch, ma il valore di uscita del flip-flop resta il precedente. Quando $CLK=1$, il latch master diventa opaco, quindi conserva il valore precedente nel nodo, indipendentemente dal dato, e il latch slave diventa trasparente, e trasporta il valore presente nel nodo all'uscita del flip-flop.

In altre parole un flip-flop D copia il dato D su Q al fronte di salita del clock, e ricorda il suo stato in tutti gli altri casi.

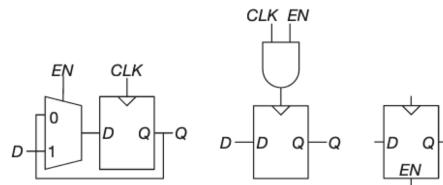


Registro

Un registro a N bit è un banco di N flip-flop che condividono un ingresso CLK comune, in modo che tutti i bit vengano aggiornati allo stesso tempo. Può anche essere realizzato con bus a N bit in ingresso e in uscita.

Flip-flop con abilitazione

Un flip-flop di questo tipo aggiunge un ingresso chiamato **EN (enable)** per determinare se memorizzare o no il dato sul fronte di salita del clock; ovvero quando è FALSO, il flip-flop ignora il clock e mantiene il proprio stato, altrimenti funziona come un normale flip-flop. In generale esistono diversi metodi per inserire un'abilitazione in un flip flop, ma in generale è preferibile non utilizzare logica combinatoria sul segnale di clock, perché questo produce un ritardo e può causare errori di temporizzazione.



Flip-flop resettabile

Anche in questo caso si aggiunge un ulteriore valore di ingresso chiamato **reset**; quando questo segnale è VERO il flip-flop resetta l'uscita a 0; altrimenti funziona come un normale flip-flop. Questi flip-flop possono essere resettabili in modo **sincrono** o **asincrono**. Quelli che funzionano in maniera sincrona si resettano al fronte di salita del clock (per realizzarli basta aggiungere una porta AND tra il dato e il reset negato), quelli in maniera asincrona nel momento in cui reset diventa VERO, indipendentemente dal valore del clock.

In generale le reti sequenziali includono tutte le reti che non sono combinatorie; esse possono contenere percorsi ciclici, nei quali le uscite sono collegate direttamente in retroazione agli ingressi. Questo tipo di reti può presentare comportamenti instabili o condizioni di corsa, ovvero in cui il corretto funzionamento della rete dipende unicamente dal ritardo delle porte che vengono utilizzate, e che quindi potrebbe creare problemi se si usa reti di produzione differente.

Per questo motivo si preferisce interrompere i percorsi ciclici inserendo in alcuni punti dei registri; questa operazione trasforma la rete in un sistema di logica combinatoria e registri. Questo perché i registri contengono lo stato del sistema, che cambia in corrispondenza dei fronti di clock (quindi lo stato è sincronizzato con il clock) e se il clock è abbastanza lento da far sì che tutti gli ingressi abbiano il tempo di adeguare il proprio valore prima del fronte del clock successivo, tutte le corse vengono eliminate.

Definiamo quindi una **rete sequenziale sincrona** come una rete che possiede le seguenti caratteristiche: una serie finita di stati discreti; un ingresso clock i cui fronti di salita indicano una sequenza di istanti di tempo nei quali hanno luogo le transizioni di stato; la specifica funzionale, che descrive lo stato prossimo del sistema e il valore di ogni sua uscita per ogni possibile combinazione di stato presente e valori d'ingresso; una specifica temporale, che consiste in un limite di tempo del fronte di salita del clock fino ai cambiamenti delle uscite.

Le regole di composizioni delle reti sequenziali sincrone sono:

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

Le reti sequenziali che non sono sincrone sono dette asincrone.

Le reti sequenziali sincrone possono essere rappresentate con le **macchine a stati finiti**; il loro nome deriva dal fatto che una rete con K registri può trovarsi in uno di un numero finito di stati diversi, nello specifico 2^K . Una FSM è composta da due blocchi di logica combinatoria, la logica di stato prossimo e la logica d'uscita, e da un registro che immagazzina lo stato. A ogni fronte di salita del clock la FSM avanza al prossimo stato, definito in base agli ingressi e allo stato presente.

Esistono due classi di FSM: le macchine alla **Moore**, in cui le uscite dipendono esclusivamente dallo stato presente della macchina (per questo i loro valori vengono indicati nei cerchi che rappresentano gli stati), e le macchine alla **Mealy**, in cui le uscite dipendono sia dallo stato presente che dagli ingressi attuali (quindi i valori delle uscite non si trovano più nei cerchi, bensì sugli archi).

Spesso è più semplice progettare FSM più complesse se queste possono essere decomposte in diverse macchine a stati più semplici che interagiscono fra loro, facendo sì che le uscite di alcune siano gli ingressi di altre; questo processo prende il nome di fattorizzazione.

Una macchina fotografica è caratterizzata da un **tempo di apertura** in cui l'oggetto che si sta fotografando deve essere fermo affinché l'immagine risulti nitida; allo stesso modo affinché una rete interpreti in maniera corretta gli ingressi, questi devono essersi stabilizzati prima del fronte di salita, quindi per il **tempo di setup**, e devono rimanere stabili dopo il fronte di salita, ovvero per il **tempo di hold**, altrimenti il comportamento della rete potrebbe essere imprevedibile.

Quando il clock presenta il fronte di salita, l'uscita inizia a cambiare dopo il ritardo di contaminazione del clock e deve stabilizzarsi sul valore definitivo entro il ritardo di propagazione del clock. Quindi il periodo del clock (tempo tra i fronti di salita) deve essere abbastanza lungo da permettere a tutti i segnali di stabilizzarsi, il che costituisce un limite per la velocità del sistema.

Definiamo **token** un gruppo di ingressi che vengono elaborati per produrre un gruppo di uscite. La **latenza** di un sistema è il tempo richiesto a un token per attraversare il sistema dall'inizio alla fine. La **capacità produttiva** è il numero di token che possono essere elaborati per unità di tempo. Per aumentare la capacità produttiva si possono elaborare più token allo stesso tempo; questo modo di lavorare viene chiamato parallelismo e può essere di due tipi:

Con il **parallelismo spaziale** vengono usate più copie dello stesso hardware in modo che più lavori possano essere svolti contemporaneamente; con il **parallelismo temporale** (anche detto pipelining) ogni compito viene diviso in fasi; in questo modo pure se ogni compito deve attraversare tutte le fasi, compiti diversi possono trovarsi in ogni fase in qualsiasi momento, cosicché compiti diversi si sovrappongono.

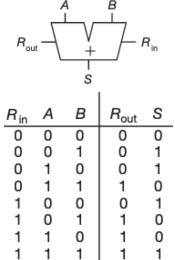
Il problema principale del parallelismo è dato dalle **dipendenze**: se un'azione dipende da un'azione precedente, tale azione non può iniziare finché quella prima non si è conclusa.

QUINTO CAPITOLO

Introduciamo adesso nuovi blocchi costruttivi sia combinatori sia sequenziali più complessi utilizzati nei sistemi digitali. I **circuiti aritmetici** sono blocchi costruttivi centrali dei calcolatori.

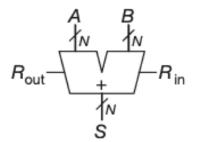
Half adder e full adder

Un semisommatore a 1 bit è costituito da due ingressi e due uscite, una che rappresenta la somma degli ingressi e l'altra che rappresenta un eventuale riporto. In un sommatore a più bit, R_{out} viene sommato al bit più significativo successivo, però al semisommatore manca un ingresso R_{in} per accettare R_{out} della colonna precedente, cosa che è invece presente in un sommatore completo, che però è sempre un sommatore a un bit.

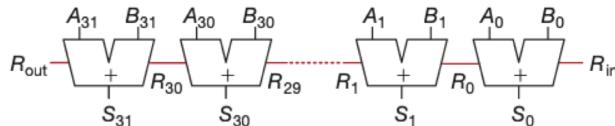


Sommatore a propagazione di riporto

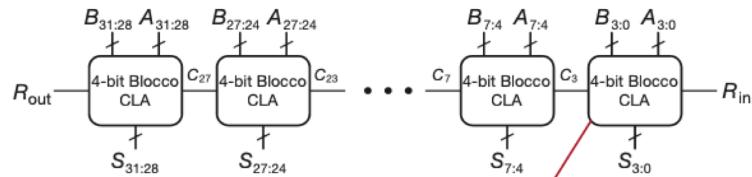
Un sommatore a N bit somma due ingressi a N bit e aggiunge R_{in} per produrre un risultato a N bit e un riporto. Questo sommatore viene comunemente chiamato **sommatore a propagazione di riporto** (o carry propagate adder-CPA) perché il riporto di un bit si propaga nel bit successivo. Le tre realizzazioni più comuni di CPA sono:



- i **ripple carry**, o sommatore a propagazione di riporto a onda, in cui per sommare due ingressi a N bit sono collegati in cascata N full adder completi, in modo che l' R_{out} di uno stadio costituisce l' R_{in} per lo stadio successivo. Il principale svantaggio di questo sommatore è il progressivo rallentamento all'aumentare dei bit.



- i **carry-lookahead adder**, o sommatore ad anticipazione di riporto, risolvono il problema della velocità dividendo il sommatore stesso in blocchi e aggiungendo un circuito per determinare velocemente il riporto di uscita da ciascun blocco appena è noto il riporto di ingresso. Questo tipo di sommatore utilizza segnali di **generazione** e di **propagazione** che descrivono come una colonna o un blocco determinano il loro riporto. Si dice che un blocco genera un riporto se questo produce un riporto indipendentemente dal valore del riporto di ingresso al blocco. Si dice invece che il blocco propaga un riporto se produce un riporto ogni qualvolta sia presente un rapporto di ingresso nel blocco. Questo metodo risulta sicuramente più veloce del precedente, ma il ritardo aumenta comunque all'aumentare dei bit.



Sottrattore

I sommatori sono in grado di sommare sia numeri positivi che negativi utilizzando la rappresentazione in complemento a due; per eseguire una sottrazione quindi basta invertire il segno del secondo numero e poi eseguire la somma, ovvero negare tutti i bit del secondo numero e inserire $R_{in}=1$.

Comparatori

Un comparatore determina se due numeri sono uguali o se uno dei due è maggiore dell'altro. Esistono due tipi di comparatori comuni:

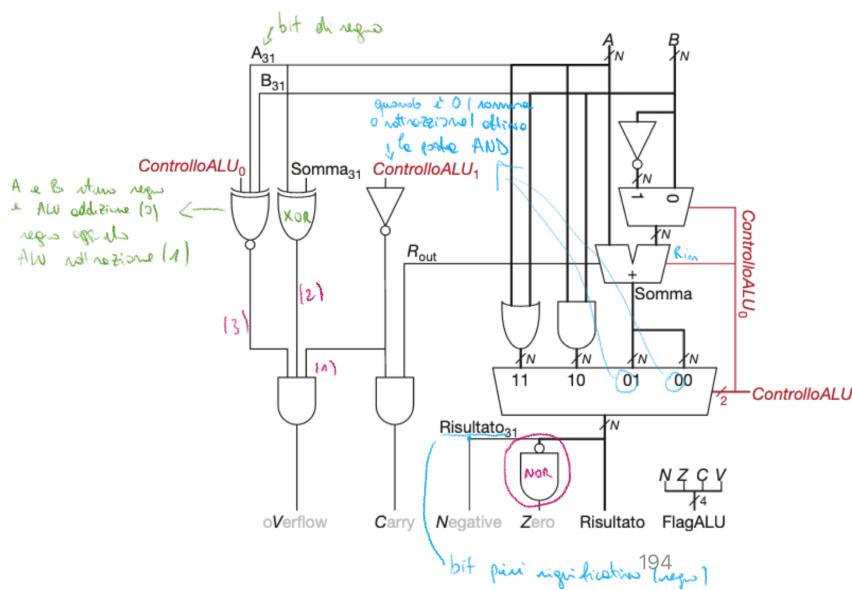
- un **comparatore di uguaglianza** produce una singola uscita che indica se i due numeri sono uguali oppure no; solitamente è più semplice a livello di hardware, perché basta utilizzare delle porte XNOR per vedere se i bit dei numeri corrispondono;
- un **comparatore di valore** produce invece una o più uscite che indicano i valori relativi di A e B. La comparazione viene di solito eseguita eseguendo la differenza tra i due ingressi e guardando al segno: se il risultato è negativo $A < B$, se è positivo $A > B$, altrimenti sono uguali. Tuttavia questo comparatore non lavora correttamente in caso di overflow.

ALU (arithmetic/logical unit)

Un'unità logica/aritmetica unisce all'interno di una singola unità una serie di operazioni logiche e matematiche, ad esempio una ALU tipica è in grado di eseguire le operazioni di addizione, sottrazione, AND e OR logici bit a bit. La ALU riceve due ingressi e un'uscita a N bit, e inoltre riceve un segnale di controllo a 2 bit, chiamato **controllo ALU**, che specifica quale funzione debba eseguire.

ControlloALU _{1:0}	Funzione
00	Addizione
01	Sottrazione
10	AND
11	OR

Un ALU è composta di: un sommatore a N bit, un numero N di porte AND e OR a due ingressi, dei negatori, un multiplexer per invertire il segno di B e un multiplexer che sceglie l'operazione desiderata sulla base del controllo. Alcune ALU producono uscite ulteriori, chiamate **flag** che danno informazioni aggiuntive sul risultato dell'ALU. In particolare queste sono N , Z , C e V , e indicano rispettivamente se il risultato è Negativo, uguale a Zero, se il sommatore ha generato un riporto (Carry) o un overflow. Affinché l'ALU possa riportare queste flag, deve controllare il bit più significativo del risultato per sapere se è negativo, vedere se tutti i bit sono nulli per sapere se è nullo, utilizza poi una porta AND per segnalare il riporto in caso questo sia presente e in caso l'ALU stia eseguendo una somma, e infine sappiamo che un overflow si verifica nelle seguenti condizioni: (1) la ALU sta eseguendo una somma o una sottrazione; (2) A e il risultato hanno segno opposto; (3) se si tratta di un'addizione A e B hanno lo stesso segno, se è una sottrazione hanno segno opposto.



Traslatori

I traslatori, o **shifter**, e i rotatori, o **rotator**, trasano i bit eseguendo la moltiplicazione o la divisione per potenze di 2. Come suggerisce stesso il nome, i traslatori trasano un numero binario a destra o a sinistra di uno specifico numero di bit e ne esistono diversi tipi:

- il **traslatore logico** trasla un numero e riempie gli spazi lasciati vuoti con 0;
- il **traslatore aritmetico**, lavora come quello logico, ma quando trasla un numero verso destra

riempie i bit più significativi con una copia del precedente bit più significativo;

Un rotatore trasla un numero verso destra o verso sinistra circolarmente, in modo che gli spazi lasciati vuoti vengano riempiti dai bit all'estremità opposta del numero.

Una traslazione a sinistra di N bit indica una moltiplicazione del numero per 2^N , mentre una traslazione a destra ne indica la divisione.

Contatori

Un contatore binario a N bit è una rete sequenziale aritmetica con ingressi di clock e di reset e un'uscita a N bit. Il reset inizializza l'uscita a 0, e il contatore genera tutti i possibili 2^N valori di uscita in ordine binario crescente, aumentando di uno a ogni fronte di salita del clock.

Registri a scorrimento

I registri a scorrimento possono essere visti come **convertitori serie-parallelo**: l'ingresso viene ricevuto in serie, ovvero un bit per volta, e dopo N cicli, gli N ingressi ricevuti sono disponibili in parallelo su Q. In particolare a ogni fronte di salita del clock il nuovo bit inserito all'ingresso e tutti i bit seguenti vengono traslati in avanti, rendendo l'ultimo bit nel registro disponibile all'uscita S_{out} .

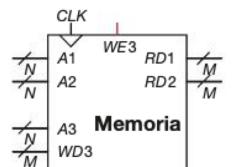
Componenti di memoria

I circuiti aritmetici e i circuiti sequenziali elaborano dati, ma i sistemi digitali richiedono anche delle memorie per immagazzinare i dati. I registri costituiti con i flip-flop sono un esempio di memoria in grado di immagazzinare piccole quantità di dati.

La memoria è organizzata come una matrice bidimensionale di celle di memoria. A ogni accesso la memoria può leggere o scrivere il contenuto di una riga di matrice, che prende il nome di **indirizzo**. Il valore letto o scritto nella memoria viene chiamato **dato**. Un componente con un numero N di bit di indirizzo e un numero M di bit di dato, possiede 2^N righe e M colonne. Ogni riga di dati viene chiamata **parola**, quindi tale componente contiene 2^N parole da M bit.

Ogni componente di memoria è realizzato come matrice di celle di bit, ognuna delle quali può contenere un bit di dato ed è connessa a una **linea di parola** e a una **linea di bit**. Durante la lettura di memoria le linee di bit sono inizialmente lasciate a un valore fluttuante, la linea di parola corrispondente all'indirizzo è attivata e la riga corrispondente di celle di bit porta le linee di bit a un valore ALTO o BASSO. Durante la scrittura invece per prima cosa vengono portate le linee di bit a un valore ALTO o BASSO, e solo successivamente viene attivata la linea di parola, permettendo così ai valori delle linee di bit di essere immagazzinati in quella riga di celle di bit.

Tutte le memorie possiedono una o più **porte**; ognuna di queste fornisce un accesso in lettura e/o in scrittura a un indirizzo di memoria. Le memorie multi-porta possono accedere a più indirizzi nello stesso momento: nell'esempio la porta 1 legge il dato all'indirizzo A1 sull'uscita di lettura dati RD1, e così la porta 2 per il dato all'indirizzo A2 su RD2, mentre la porta 3 scrive il dato dall'ingresso di scrittura dati WD3 all'indirizzo A3 sul fronte di salita del clock se l'abilitazione di scrittura WE3 è abilitata.



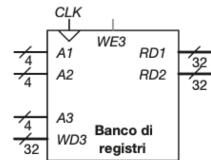
Le memorie vengono classificate in base a come immagazzinano i bit nella cella di bit. Distinguiamo principalmente le **memorie ad accesso casuale (RAM)** e le **memorie a sola lettura (ROM)**. La RAM è una memoria **volatile**, cioè una memoria che perde traccia dei suoi dati una volta spenta, mentre la ROM è una memoria **non volatile**, cioè una memoria che trattiene i suoi dati per un tempo indefinito, anche in assenza di alimentazione.

I due tipi principali di memorie RAM sono la **RAM dinamica (DRAM)**, e la **RAM statica (SRAM)**.

La DRAM memorizza un bit come presenza o assenza di una carica in un condensatore; quando la linea di parola è attiva, il transistore si accende e il valore del bit immagazzinato viene trasferito alle o dalla linea di bit. In caso di lettura, il valore di dato viene trasferito dal condensatore alla linea di bit, ovvero la lettura distrugge il valore del bit immagazzinato nel condensatore, quindi la parola di dato deve essere rinfrescata dopo ogni lettura. Anche quando la DRAM non viene letta è necessario rinfrescare i contenuti, cioè leggerli e riscriverli, ogni pochi millisecondi poiché la carica sul condensatore si perde gradualmente.

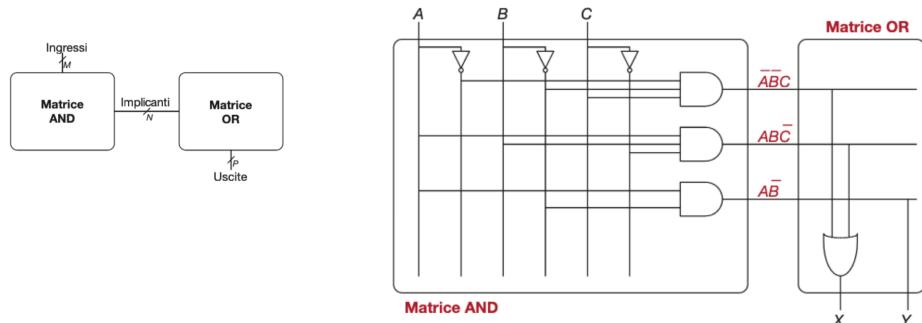
La SRAM viene chiamata così perché i bit immagazzinati nella memoria non hanno bisogno di essere ricaricati, questo perché i bit di dati vengono immagazzinati grazie a dei negatori collegati a croce, quindi se il rumore deteriora il valore del bit, i negatori collegati a croce lo riportano al valore di partenza. Queste caratteristiche la rendono più veloce di una DRAM, ma anche più costosa.

I sistemi digitali utilizzano spesso un certo numero di registri per immagazzinare variabili temporanee. Questo gruppo di registri, chiamato **banco di registri**, viene solitamente costruito come una piccola componente SRAM multi-porta, perché questa è più compatta rispetto a una matrice di flip-flop.



Una ROM memorizza un bit come presenza o assenza di un transistore. Si noti che quindi una cella di bit di una ROM è una rete combinatoria e non ha uno stato che le consenta di dimenticare il proprio contenuto quando viene spenta i contenuti delle celle della ROM vengono specificati durante la sua costruzione dalla presenza o assenza di un transistore; le ROM programmabili hanno un transistore per ogni cella di bit che consente di collegare o scollegare un transistore da massa; le ROM riprogrammabili possiedono invece un meccanismo reversibile per connettere o disconnettere il transistore.

La forma SOP viene chiamata **logica a due livelli** perché consiste di letterali connessi a un primo livello di porte AND che, a loro volta, sono connesse a un secondo livello di porte OR. Così una qualsiasi espressione booleana in forma somma di prodotti può essere tradotta in schema circuitale (diagramma di una rete digitale) in maniera sistematica; questo stile di disegno è chiamato **matrice logica programmabile (PLA)**, perché i negatori e le porte logiche sono allineati in maniera sistematica. Queste matrici sono dunque costituite da una matrice AND seguita da una matrice OR; ogni riga della matrice AND genera un implice e i punti nella matrice OR indicano invece quali implicanti fanno parte di ciascuna funzione di uscita. Una ROM può essere vista come un caso particolare di PLA. Questo tipo di matrici è però stato col tempo rimpiazzato dalle FPGA, ovvero matrici di porte logiche programmabili sul campo, perché hanno i vantaggi di essere riconfigurabili, di poter realizzare sia funzioni combinatorie che sequenziali, e di poter realizzare funzioni logiche a più livelli.



SESTO CAPITOLO

Definiamo l'**architettura** di un elaboratore in termini di set di istruzioni (il linguaggio) e di locazione degli operandi (registri e memoria). Il primo passo per comprendere un'architettura è conoscerne il linguaggio: le parole che lo costituiscono sono chiamate **istruzioni**, e il vocabolario delle possibili parole è il set di istruzioni. Le istruzioni indicano sia l'operazione da eseguire che gli operandi da utilizzare, e devono essere codificate come stringhe binarie in quello che si definisce linguaggio macchina. L'architettura ARM rappresenta ogni istruzione con una parola di 32 bit, ma per renderle più comprensibili all'uomo si preferisce rappresentare le istruzioni in forma simbolica chiamata linguaggio assembly. L'architettura di un elaboratore non definisce la sottostante struttura circuitale: possono esistere diverse microarchitetture per la stessa architettura.

La prima parte dell'istruzione assembly è chiamata **mnemonico** e indica l'operazione da eseguire; tale operazione deve essere eseguita sugli **operandi sorgente**, che seguono l'**operando destinazione**, dove sarà riportato il risultato. Le istruzioni richiedono quindi locazioni fisiche dalle quali prelevare i dati binari; gli

operandi possono essere memorizzati nei registri del processore, o in memoria, o possono essere costanti (**immediati**, è preceduto dal carattere #) memorizzate nelle stesse istruzioni.

Le istruzioni hanno bisogno di raggiungere velocemente gli operandi per poter essere eseguite rapidamente, ma gli operandi salvati in memoria richiedono tempi lunghi per essere recuperati. Per questo motivo quasi tutte le architetture definiscono un limitato numero di registri per memorizzare gli operandi più usati; meno sono i registri, più rapidamente sono accessibili.

Nome	Utilizzo
R0	Parametro/valore da restituire/va-
R1-R3	Parametri/variabili temporanee
R4-R11	Variabili salvate
R12	Variabile temporanea
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

L'istruzione **MOV** è molto comoda per inizializzare i registri, e può usare sia immediati che registri come operandi sorgente (MOV R4, #0x2A oppure MOV R5, R7).

Ma i dati possono essere memorizzati anche nella memoria di lavoro che è più grande e più lenta, ma in ARM le istruzioni operano solo sui registri, quindi i dati in memoria devono essere copiati nei registri prima di poter essere elaborati.

ARM fornisce l'istruzione **LDR** (load register) per leggere una parola di dato dalla memoria in un registro (LDR R7, [R5, #8] ovvero R7=dato in indirizzo R5+8). Il numero tra parentesi è l'**indice** di parola; l'istruzione LDR specifica l'indirizzo di memoria usando un **registro di base** e un **offset**, non sempre necessario. Per emulare la struttura degli **array**, ARM può moltiplicare l'indice, sommarlo all'indirizzo base e leggere il dato da memoria in un'unica istruzione: LDR R3, [R0, R1, LSL #2].

ARM usa l'istruzione **STR** (store register) per scrivere una parola di dato da un registro in memoria (STR R9, [R1, #0x14] salva il valore del registro nella cella di indirizzo R1+20).

Modo	Assembly ARM	Indirizzo	Registro base
Offset	LDR R0, [R1, R2]	R1 + R2	non modificato
Pre-indicizzato	LDR R0, [R1, R2]!	R1 + R2	R1 = R1 + R2
Post-indicizzato	LDR R0, [R1], R2	R1	R1 = R1 + R2

Le istruzioni logiche includono AND, ORR, EOR (XOR) e BIC (bit clear). Tutte operano bit a bit e scrivono il risultato in un registro destinazione; la prima sorgente è sempre un registro e la seconda può essere un registro oppure un immediato.

Le istruzioni di traslazione (shift) in ARM sono LSL (logical shift left), LSR (logical shift right), ASR (arithmetic shift right) e ROR (rotate right). Queste operazioni sono già state descritte precedentemente, ma è interessante notare che non esiste una rotazione a sinistra perché questa può essere realizzata con una rotazione a destra di un numero di passi complementare. L'ampiezza della traslazione può essere un immediato o un registro.

L'istruzione **MUL** moltiplica due valori a 32 bit e produce un risultato a sua volta a 32 bit. Le istruzioni UMULL (unsigned multiply long) e SMULL (signed multiply long) moltiplicano due valori a 32 bit e producono un risultato a 64 bit.

Le istruzioni ARM possono opzionalmente impostare a 0 o a 1 delle flag di condizione, dette anche flag di stato (N, Z, C, V). Il modo tipico di impostarle è tramite l'istruzione di confronto **CMP** (compare), che sottrae il secondo operando sorgente al primo e imposta le flag sulla base del risultato. Lo mnemonico delle istruzioni successive può essere seguito da un mnemonico di condizione che indica quando eseguirla. Altre istruzioni di elaborazione dati impostano le flag quando lo mnemonico dell'istruzione è seguito da "S".

ARM usa **istruzioni di salto** (branch instructions) per saltare parti di codice o ripeterle. Un programma viene normalmente eseguito in sequenza con il registro **PC (program counter)** che si incrementa di 4 dopo ogni istruzione per puntare alla successiva; questo perché in ARM sia le istruzioni che le parole di dato sono di 32 bit (4 byte), e ARM fa uso di una memoria *byte addressable*, cioè ogni byte di memoria ha un indirizzo univoco. Quindi ogni indirizzo di parola è multiplo di 4. Le istruzioni di salto modificano il PC: quando si raggiunge l'istruzione **B LABEL**, la prossima istruzione che viene effettuata è quella che segue l'etichetta. Come altre istruzioni ARM, i salti possono essere incondizionati o condizionati.

Il costrutto *if* esegue un blocco di codice solo se una certa condizione è verificata; il codice assembly valuta la condizione opposta, e salta il blocco di codice se questa non è verificata. In generale, quando un blocco di codice è costituito da una sola istruzione, conviene usare l'esecuzione condizionata. Per il costrutto *if else*, se la condizione non è verificata, si salta il blocco *if* per eseguire quello *else*, altrimenti si procede nel blocco *if*, il quale termina con un salto incondizionato per saltare il blocco *else*.

Per il blocco *while*, si controlla sempre la condizione opposta, e se questa è verificata si esce dal ciclo, altrimenti si continua con il blocco del ciclo che termina con un salto incondizionato all'inizio di esso dove avviene la nuova verifica. Allo stesso modo si costruisce il ciclo *for*.

I linguaggi di alto livello supportano i **sottoprogrammi** per consentire di riutilizzare parti comuni di codice e rendere i programmi più leggibili. In particolare, le funzioni ricevono valori di ingresso, denominati parametri, e forniscono un solo valore di uscita, denominato valore di ritorno.

In ARM convenzionalmente il chiamante mette fino a quattro parametri nei registri R0-R3 prima di eseguire la chiamata a funzione, e il chiamato mette il valore di ritorno in R0 prima di terminare. Il chiamato non deve modificare nessuno dei registri o delle celle di memoria necessari al chiamante, in particolare deve lasciare inalterati i registri preservati (R4-R11, LR) e lo **stack**, ovvero la porzione di memoria usata dal chiamante per le proprie variabili temporanee. Il chiamante memorizza l'indirizzo di ritorno nel registro **LR (link register)** nel momento in cui salta a eseguire il chiamato con l'istruzione **BL LABEL** (branch and link), e copia il contenuto di tale registro nel PC (MOV PC, LR) per ritornare al chiamante. Se è necessario chiamare una funzione con più di quattro parametri, i parametri aggiuntivi vanno messi nello stack.

Lo stack è la parte di memoria usata per salvare informazioni all'interno di una funzione; si espande quando il processore nell'esecuzione del programma ha bisogno di più spazio, e si contrae liberando memoria quando il processore non ne ha più bisogno. Lo stack è una coda del tipo *last in first out*.

Lo **stack pointer SP** (R13) è un normale registro, che però per convenzione punta alla cima dello stack, ed è lo strumento utilizzato per allocare e deallocare memoria. Una funzione deve allocare spazio nello stack, salvare i dati dei registri prima di modificarli, e ripristinare i valori originari, e infine deallocare lo spazio prima di terminare.

Le istruzioni **PUSH** e **POP** sono il modo preferenziale di salvare e ripristinare registri nello stack pieno discendente convenzionale. Sicuramente R0 non è preservato perché conserva il valore di ritorno; mentre LR deve essere sempre salvato perché contiene l'indirizzo di ritorno della funzione.

Una funzione che non chiama un'altra funzione viene definita **funzione foglia**; una funzione che al suo interno chiama un'altra funzione viene quindi definita **funzione non-foglia**. Per quest'ultimo tipo valgono le seguenti regole: il chiamante, prima di chiamare la funzione, deve salvare ogni registro non preservato (R0-R3 e R12) che intende usare dopo la chiamata, e successivamente ripristinarlo prima di utilizzarlo, mentre il chiamato, prima di modificare un registro preservato (R4-R11 e LR), deve salvare il contenuto e ripristinarlo prima di ritornare al chiamante.

Una **funzione ricorsiva** è una funzione non-foglia che chiama se stessa. Si comporta quindi sia da chiamante che da chiamato, e deve pertanto salvare sia i registri preservati sia quelli non preservati.

Il linguaggio assembly è abbastanza comodo per essere interpretato dall'uomo, ma il calcolatore lavora con istruzioni scritte in **linguaggio macchina**, ovvero sotto forma di stringhe di 32 bit di 0 e di 1. In ARM ci sono tre principali tipi di istruzioni: elaborazione dati, accesso a memoria e salti.



Istruzioni di elaborazione dati

Questo tipo di istruzioni ha come primo operando sorgente un registro e un secondo che può essere un immediato o un altro registro, eventualmente traslato; un terzo registro è la destinazione. L'istruzione a 32 bit ha sei campi:

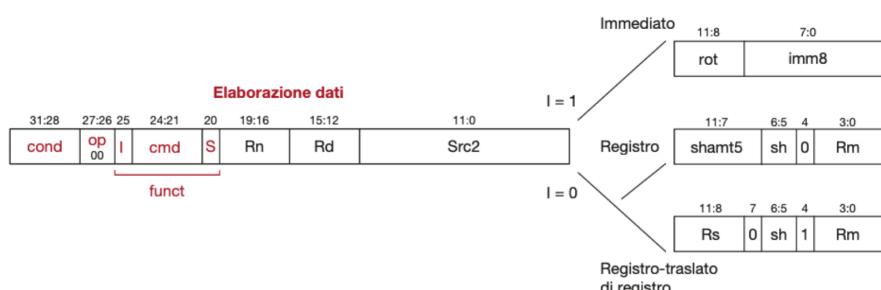
L'operazione che l'istruzione deve eseguire è codificata nei campi *op*, che per le elaborazioni dei dati è 00, e *funct*; il campo *cond* codifica l'eventuale esecuzione condizionata, ed è 1110 per le istruzioni non condizionate. Infine Rn è il registro del primo operando sorgente, Rd è il registro di destinazione, e Src2 è il secondo operando sorgente.

Funct ha tre sottocampi: il bit *I* vale 1 quando Src2 è un immediato, il bit *S* vale 1 quando l'istruzione impone la flag di condizione, e *cmd* indica il tipo di operazione da eseguire.

Src2 ha tre varianti che consentono di avere come secondo operando: un immediato, che viene rappresentato con 8 bit traslati eventualmente del numero indicato nei 4 bit di *rot* moltiplicato per 2;

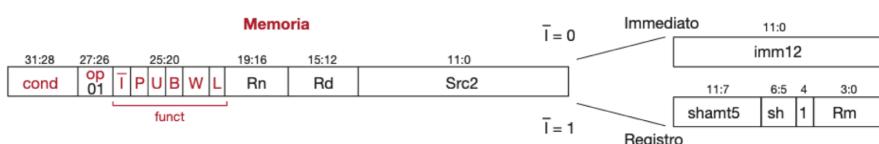
un registro eventualmente traslato di una costante per le operazioni di shift, per cui 2 bit di *sh* sono usati per definire di quale tipo di traslazione si tratta, Rm contiene il registro da traslare, *shamt5* specifica il valore di traslazione (nota che Rn non è utilizzato, quindi viene lasciato a 0);

un registro traslato del contenuto di un altro registro, per cui i bit dedicati precedentemente al valore di traslazione, sono utilizzati per indicare il registro contenente il valore di traslazione.



Istruzioni di accesso a memoria

Queste istruzioni utilizzano un formato simile a quello precedente con Rd il registro sorgente per lo store e di destinazione per il load, Rn il registro di base, con la differenza che hanno una differente configurazione del campo funct, due variabili del campo Src2, che in questo caso rappresenta l'offset, e il campo op=01. L'offset può essere un immediato unsigned a 12 bit, oppure un registro eventualmente traslato di una costante. Funct è costituito da 6 bit di controllo: *T** e *U* che segnalano se l'offset è un immediato o un registro, e se deve essere sommato o sottratto; *P* e *W* che definiscono il modo in cui viene gestito l'indice (preindividuazione, ecc); *L* e *B* specificano il tipo di accesso alla memoria.



Istruzioni di salto

Le istruzioni di salto utilizzano un unico immediato signed a 24 bit, 4 bit per l'istruzione condizionata, op=10, e il campo funct è ridotto a due bit: il primo sempre uguale a 1 e l'altro che segnala se l'istruzione è un branch o un branch e link. Il campo immediato contiene il numero di istruzioni comprese tra l'istruzione da raggiungere e PC+8. Per calcolare quindi dove arriva il salto, basta estendere l'immediato a 32 bit, traslarlo a sinistra di due posizioni, e sommarlo a PC+8.



SETTIMO CAPITOLO

La **microarchitettura** consiste nella specifica combinazione di registri, ALU, macchine a stati finiti, memorie e altri blocchi logici necessari per la realizzazione di un'architettura. Un'architettura può avere molte diverse microarchitettura, ma tutte devono essere in grado di eseguire gli stessi programmi; ne vediamo in particolare tre.

È opportuno dividere la microarchitettura in due parti tra loro interagenti: il **datapath** e l'**unità di controllo**. Il percorso dati opera su parole di dati; è costituito da strutture come memorie, registri, ALU e multiplexer. L'unità di controllo riceve l'istruzione corrente del percorso dati e comunica al percorso dati come eseguirla, attivando opportunamente gli ingressi di selezione del multiplexer, le abilitazioni dei registri e i segnali di lettura e scrittura in memoria.

Microarchitettura a ciclo singolo

Questo tipo di microarchitettura esegue un'intera istruzione in un ciclo di clock, il cui tempo è impostato dall'istruzione più lunga; inoltre il processore richiede una memoria istruzioni e una memoria dati separate, situazione generalmente non realistica.

I componenti principali sono il program counter, il banco di registri di lavoro, il registro di stato, le memorie istruzioni e dati. Anche se il PC fa concettualmente parte dei registri, viene letto e scritto in ogni ciclo di clock indipendentemente dalle normali operazioni degli altri registri, quindi conviene realizzarlo come registro autonomo.

L'unità di controllo genera i segnali di controllo sulla base dei campi dell'istruzione, delle flag dell'ALU e del fatto che il registro di destinazione sia il PC. Deve anche memorizzare e aggiornare opportunamente le flag di stato. In generale è costituita da un decoder, che genera la maggior parte delle abilitazioni, e da una logica condizionale che mantiene le flag di stato e segnala quando un'operazione deve essere eseguita in maniera condizionata. Più nello specifico il decoder è composto di: un decoder principale che genera la maggior parte dei segnali, un decoder dedicato all'ALU che usa il campo funct per determinare l'operazione, e da una logica del PC per decidere se il PC deve essere modificato.

Questo tipo di architettura ha tre principali svantaggi: richiede memorie separate per istruzioni e dati, mentre la maggior parte dei processori ha un'unica memoria esterna, richiede un ciclo di clock più lungo di quanto la maggior parte delle istruzioni abbiano bisogno, e richiede tre circuiti sommatori, alquanto costosi.

Microarchitettura multi ciclo

Questa microarchitettura esegue le istruzioni in sequenze di cicli più brevi, in modo che le operazioni più brevi vengano completate in meno cicli di quelle più complesse; inoltre riduce il costo hardware riutilizzando blocchi circuituali costosi come i sommatori e le memorie. Riesce a ottenere questo risultato aggiungendo vari registri non architettonici per memorizzare valori intermedi.

Visto che l'unità di controllo deve generare segnali di controllo diversi nei diversi passi di esecuzione di una singola istruzione, serve una macchina a stati finiti, e non più la logica combinatoria del processore a ciclo singolo. Quindi se andiamo ad analizzare la sua struttura, questa è sempre divisa in un decoder e in una parte di logica combinatoria, ma il decoder è formato da: un decoder per l'ALU, la logica del PC, una FSM principale che sostituisce il decoder principale del ciclo singolo, di tipo Moore affinché le uscite dipendano unicamente

dallo stato corrente; ma alcuni segnali dipendono invece da op, invece che dallo stato corrente, per questo viene aggiunto un piccolo decoder istruzioni per gestire tali segnali.

In realtà, nonostante questo tipo di microarchitettura sia stata pensata per sopperire ai difetti di quella a ciclo singolo, facendo un'analisi delle prestazioni si scopre che il processore è più lento di quello a ciclo singolo a causa delle latenze di propagazione. In generale ha lo svantaggio di poter eseguire una sola istruzione per volta e di essere abbastanza lento, ma ha bisogno di una sola memoria e di meno componenti in generale, il ché lo rende più economico, per questo viene impiegato spesso in sistemi a basso costo.

Microarchitettura pipeline

Può eseguire più istruzioni per volta, migliorando sensibilmente le prestazioni; serve però aggiungere una logica per gestire le dipendenze tra le istruzioni in esecuzione simultaneamente. I processori pipeline devono poter accedere a istruzioni e dati nello stesso ciclo, e a questo scopo usano generalmente memorie cache separate per istruzioni e dati.

Si progetta un processore pipeline suddividendo il processore a ciclo singolo in cinque stadi di pipeline: **fetch**, in cui il processore legge l'istruzione dalla memoria, **decode**, in cui legge gli operandi dal banco di registri e decodifica l'istruzione per generare i segnali di controllo appropriati, **execute**, in cui si eseguono i calcoli con l'ALU, **memory**, in cui legge o scrive dati in memoria, e **writeback**, in cui scrive il risultato del banco di registri, quando previsto dall'istruzione. Visto che il processore pipeline utilizza gli stessi segnali di controllo del processore a ciclo singolo, ha anche la stessa unità di controllo. In questo modo cinque istruzioni alla volta possono essere in esecuzione, una per ogni stadio; in questo modo il throughput, ovvero la capacità di lavoro, dovrebbe essere idealmente cinque volte superiore.

In realtà l'adozione di una struttura pipeline richiede attività aggiuntive, per cui la capacità di lavoro non cresce come ci si potrebbe aspettare idealmente, ma si hanno comunque moltissimi vantaggi a costi così bassi.

OTTAVO CAPITOLO

Si ottengono benefici relativi ad un'ampia collezione di dati e a una rapidità di accesso, tramite una gerarchia di immagazzinamento, fondata a partire da due principi: il principio della **località temporale** che suggerisce la probabilità di dover utilizzare in futuro un dato appena recuperato, e il principio della **località spaziale**, ovvero che se si utilizza un dato, è molto probabile che si debbano utilizzare dati a esso contigui (come negli array).

I sottoinsiemi di memoria usati per costruire questa gerarchia sono le DRAM e le SRAM; idealmente un sistema di memoria deve essere veloce, grande ed economico, ma in pratica ogni tipo di memoria ha solo due di questi requisiti. Per simulare quindi una memoria perfetta idealmente si combinano generalmente una memoria veloce, economica ma piccola, con una memoria grande, economica ma lenta. La memoria veloce memorizza le istruzioni e i dati usati più frequentemente, quindi in media il sistema risulta veloce. La memoria grande memorizza il resto delle istruzioni e dei dati, quindi la capacità totale del sistema risulta elevata. Si possono estendere questi principi costruendo una gerarchia di memorie di capacità crescente e velocità decrescente.

I calcolatori memorizzano istruzioni e dati usati più frequentemente in una memoria più veloce ma più piccola chiamata **cache** e realizzata da SRAM; le cache possono memorizzare sia istruzioni che dati, ma generalmente si parla genericamente di dati. Se il processore richiede un dato che è presente nella cache si parla di **hit**; in caso contrario il processore deve recuperarlo nella memoria principale, e si parla di **miss**. Se la cache dà luogo a hit nella maggior parte dei casi, il processore deve attendere solo raramente le risposte della lenta memoria principale, e il tempo medio di accesso risulta breve.

In un tipico PC la memoria principale è costituita da una DRAM.

Il terzo livello della gerarchia di memoria è costituito da un **disco rigido**, per contenere i dati che non trovano spazio nella memoria principale. Il disco rigido dà l'illusione di una capacità di memoria molto più ampia delle dimensioni della memoria principale: si parla quindi di **memoria virtuale**. La memoria principale, detta anche memoria fisica, contiene un sottoinsieme della memoria virtuale, quindi può essere pensata come una cache dei dati del disco rigido utilizzati più frequentemente.

Quando la cache preleva una parola dalla memoria principale, preleva anche alcune altre parole adiacenti: questo gruppo di parole è denominato **blocco**. Il numero di parole per blocco di cache è definito **dimensione del blocco**; una cache con **capacità** (numero di parole che può contenere) C , contiene quindi C/b blocchi.

Ogni cache è organizzata in **set** o insiemi, ciascuno dei quali contiene uno o più blocchi di dati. La relazione tra l'indirizzo di un dato in memoria principale e la locazione di tale dato in cache è definita **mappatura**. Ogni indirizzo di memoria viene mappato in un set della cache; alcuni dei bit dell'indirizzo vengono usati per determinare in quale set della cache è contenuto il dato. Se il set contiene più di un blocco, il dato può essere memorizzato in uno qualsiasi dei blocchi del set.

In una **cache a mappatura diretta** ogni set contiene un solo blocco, quindi la cache ha $S=B$ set. Per comprendere questo tipo di mappatura si deve dividere la memoria principale in blocchi di b parole ciascuno, esattamente come nella cache. Un indirizzo nel blocco 0 della memoria viene mappato nel set 0 della cache, e così via fino a un indirizzo nel blocco $B-1$ della memoria principale che viene mappato nel set $B-1$. Non ci sono altri set nella cache, quindi la mappatura si ripete circolarmente. Quindi preso un indirizzo di memoria, questo è costituito, partendo dal bit meno significativo, da due bit di spiazzamento del byte, bit di set che indicano in quale set viene mappato il dato, e i restanti bit di tag indicano l'indirizzo del dato effettivamente contenuto in un certo set della cache.

Quando due indirizzi generati di recente dal processore si mappano nello stesso blocco, si verifica un **conflitto**, e il dato cui si accede per ultimo espelle il precedente dal blocco.

In una **cache set associativa** o parzialmente associativa a N vie, ogni set contiene N blocchi; quindi l'indirizzo in memoria principale è mappato in un solo set, con $S=B/N$ set, ma il dato corrispondente a tale indirizzo può finire in uno qualsiasi degli N blocchi. Questa organizzazione riduce i conflitti, il che porta le cache ad avere dei tassi di hit superiori, ma i multiplexer necessari a realizzarle, le rendono generalmente più lente e più costose.

Una **cache completamente associativa** ha un solo set, quindi il dato può andare in uno qualsiasi dei B blocchi. Alla richiesta di un dato, B confronti devono essere effettuati, dal momento che il dato potrebbe trovarsi in un blocco qualsiasi. Per questo, questo tipo di cache tende ad avere la percentuale di hit più alta, ma per via dell'ampia richiesta di hardware di cui hanno bisogno, sono adatte a cache di piccole dimensioni.

Per sfruttare anche la località spaziale, una cache utilizza blocchi più grandi per memorizzare più parole di memoria consecutive. Il vantaggio principale è che in caso di miss, il sistema copia in cache non solo il dato richiesto ma anche le parole adiacenti; gli accessi successivi hanno quindi una maggiore probabilità di hit. Tuttavia una maggiore dimensione del blocco, significa che la cache ha meno blocchi, quindi più probabilità di dar luogo a dei conflitti. Inoltre serve più tempo per prelevare dalla memoria principale tutte le parole del blocco e trasferirle in cache.

In caso della dimensione dei blocchi maggiore di uno, dopo i bit di spiazzamento del byte, si trovano i bit di spiazzamento del blocco per muoversi tra le locazioni di esso.

Ricordiamo che nell'architettura ARM con indirizzi a 32 bit e parole di 32 bit, la memoria è indirizzabile al byte e ogni parola è costituita da quattro byte, quindi gli ultimi due byte sono utilizzati per indicare il byte in cui si trova il dato.

Nelle cache a mappatura diretta quando si deve scrivere un nuovo dato in un set già pieno, semplicemente si rimpiazza il vecchio dato con il nuovo; con gli altri tipi di mappatura invece si deve scegliere quale blocco espellere, e in generale, affidandosi al principio di località temporale, si sceglie quello meno recentemente utilizzato. Questo viene fatto dividendo i blocchi di un set in due gruppi, e con un bit di utilizzo si segnala quale è stato usato meno recentemente. Al momento di sostituire il dato, un blocco viene scelto casualmente dal gruppo col bit attivo.

Le scritture in memoria seguono una procedura analoga: in una cache **write-through** il dato viene scritto simultaneamente sia nella memoria cache sia nella memoria principale; in una cache **write-back** un bit di modifica è associato a ogni blocco di cache e si attiva se il blocco è stato modificato da almeno una scrittura, e i dati vengono riscritti nella memoria principale solo al momento di essere espulsi dalla cache. Una memoria del primo tipo richiede chiaramente più accessi alla memoria principale, quindi è generalmente più lenta.

La memoria virtuale è suddivisa in **pagine virtuali**; la memoria fisica è anch'essa suddivisa in **pagine fisiche** della stessa dimensione. Una pagina virtuale può trovarsi in memoria fisica oppure sul disco rigido. Il processo di determinare l'indirizzo fisico a partire da quello virtuale prende il nome di traduzione dell'indirizzo. Per evitare mancanze di pagina dovute a conflitti, ogni pagina virtuale può essere mappata in qualsiasi pagina fisica, e visto che normalmente accedere al disco rigido richiede molto tempo, quando questo accade si scambiano diverse quantità di pagine. Praticamente la suddivisione delle pagine è simile all'indirizzamento per una mappatura completamente associativa, ma visto che non sarebbe possibile realizzare un hardware per confrontare tutte le pagine, si usa una tabella delle pagine per effettuare la traduzione. Questa solitamente si trova nella memoria principale, quindi a ogni istruzione di lettura o scrittura richiede due accessi alla memoria principale, uno per accedere alla tabella e tradurre l'indirizzo, e l'altro per leggere o scrivere il dato.