Appunti Calcolatori Elettronici

Bozzo Francesco Conti Samuele Daniotti Filippo

11 marzo 2019

Indice

Capitolo 1

Introduzione ai calcolatori

1.1 Un po' di storia dell'informatica

ENIAC, il primo computer della storia Commissionato nel 1946 dal Dipartimento di Difesa degli Stati Uniti, **ENIAC** (*Electonic Numerical Integrator and Computer*) diventa il primo calcolatore della storia. Dotato di 18000 valvole termoioniche, esso occupava una stanza di 9x30 metri, consumando un ammontare spropositato di energia.
Il suo impiego principale consisteva nel calcolare traiettorie dei proiettili di artiglieria. Infatti, è doveroso specificare come i primi calcolatori della storia siano stati concepiti per essere sfruttati in applicazioni belliche.

Apollo Guidance Computer Prodotto da IBM nel 1969, disponeva di 2800 circuiti integrati, un processore da 0.043 MHz e 152 KByte complessivi di memoria ROM/RAM. Presentava un'interfaccia display&keyboard: i comandi utilizzano una sintassi del tipo "verbo + nome".

Programma 101 Nel 1964 l'Olivetti rilascia il primo *personal computer* della storia, sfortunatamente non avrà successo.

1.1.1 Differenziazione dei calcolatori

Seppur i calcolatori di oggi condividano la stessa idea base, le soluzioni per ciascuna tipologia di applicazione sono piuttosto diverse. A seguire alcuni esempi di diverso tipo di calcolatori:

Tipo di calcolatore	Costo	Prestazioni	Applicazioni
Personale	Ridotto	Buone	Software prodotti da terze parti
Server	Elevato	Eccellenti	Poche ma complesse (calcolo scientifico) o tante ma semplici (web server)
Embedded	Minimo	Minime necessarie	Dedicate e molto specifiche; esiste un vasto spettro di scopi possibili

Per ovvi motivi, ogni tipologia di calcolatore è meglio adatta per differenti scopi. Ottenere delle buone prestazioni senza eccedere in prezzo e potenza è ciò che decreta il successo commerciale di un prodotto.

1.2 Le prestazioni dei calcolatori

Un buon programmatore, oltre a saper utilizzare degli efficienti paradgmi, deve comprendere la gerarchia di memoria e il concetto di parallelismo: conoscere a fondo il calcolatore è fondamentale.
Fino a qualche tempo fa le prestazioni di qualsiasi calcolatore erano in balia della disponibilità di memoria. Al giorno d'oggi invece risulta un problema risolto, tranne per qualche criticità per le applicazioni embedded.
Ecco una lista degli elementi che influenzano le prestazioni del calcolatore e il loro ruolo:

- *Algoritmi*: determinano il numero di istruzioni di alto livello e di operazioni di IO.
- *Linguaggi di programmazione, compilatori e architetture*: determinano il numero di istruzioni macchina per ogni istruzione di basso livello.
- *Processore e sistema di memoria*: determina quanto velocemente è possibile eseguire ciascuna istruzione.
- *Sistema di I/O (Hardware e sistema operativo)*: determina quanto velocemente possono essere eseguite le istruzioni.

1.2.1 La memoria e il processore

La memoria È possibile classificare la memoria in:

- *volatile*: è costituita da vari banchi di (tipicamente) 8 chip di RAM dinamica. È dominata dalle *DRAM*.

- *permanente*: è costituita da memorie flash (es. SSD), dischi rigidi e CD/DVD.

La prima viene utilizzata per memorizzare dati e programmi mentre vengono eseguiti (per questo motivo viene chiamata anche *memoria principale*): allo spegnimento i dati vengono persi. La seconda viene usata per memorizzare grandi quantità di dati e programmi fra esecuzioni diverse.
Il principio di funzionamento dell'hard disk è di magnetizzare delle particelle metalliche distribuite su un substrato:

- I dischi sono organizzati in strutture sovrapposte (cilindri).
- Le particelle vengono lette da un dispositivo meccanico (testina) che si sposta radialmente su un braccio (in grado di fare movimenti angolari).
- Questa componente rallenta i tempi di accesso ma aumenta al densità di memorizzazione (è possibile arrivare facilmente ai Terabyte).

Diversamente dai dischi rigidi elettromeccanici, la memorizzazione su una memoria flash avviene intrappolando una carica elettrica in maniera permanente.
I dischi ottici funzionano sul principio di riflessione della luce: viene emesso un raggio laser che viene riflesso dai rilievi, nel caso bit 1, e assorbito dalle buche, nel caso bit 0. Nei dischi riscrivibili è inoltre presente un particolare substrato che se riscaldato torna alla condizione di partenza, eliminando tutti i dati memorizzati.

Il processore La **CPU** è l'unità centrale di ogni calcolatore. Si compone di:

- *Datapath*: esegue operazioni aritmetiche sui dati.
- *Control Unit*: impartisce ordini al datapath, alla memoria e alle componenti IO, sulla base di quanto stabilito dal programma.

Il processore offre l'**ISA** (*Instruction Set Architecture*), un'interfaccia che permette di utilizzarlo senza conoscerne i dettagli. In aggiunta all'ulteriore interfaccia del sistema operativo, insieme formano l'interfaccia binaria delle applicazioni **ABI** (*Application Binary Interface*). Ciò permette allo sviluppatore di svincolarsi dal livello hardware sottostante, secondo il principio di *astrazione*.
Esistono due diversi tipi di architetture ISA:

- **RISC**: offre un numero ridotto di istruzioni, rivolta soprattutto a sistemi *embedded*;
- **CISC**: offre Permette di usufruire di un maggior numero di istruzioni, rivolta prevalentemente ai PC.

1.2.2 Software di sistema

Il *sistema operativo*:

- gestisce le operazioni di I/O.
- alloca la memoria.
- consente il multitasking.

Il *compilatore* traduce da linguaggio ad alto livello a linguaggio macchina. Ogni istruzione di quest'ultimo è costituita da una determinata sequenza di *bit*, ossia l'unità di base dell'informazione (1 o 0).
L'utilità del compilatore consiste nel facilitare il lavoro del programmatore: non è più necessario implementare il software in codice binario, bensì è possibile utilizzare un linguaggio di programmazione più vicino al linguaggio naturale. Il primo linguaggio sviluppato per questo scopo è stato l'*assembly*. Il software che traduce l'assembly in codice binario si chiama *assembler*.
È possibile riassumere la traduzione di un linguaggio di programmazione di alto livello in codice macchina in due semplici passi:

1. traduzione del linguaggio ad alto livello in linguaggio assembly, per via del compilatore
2. traduzione del linguaggio assembly in linguaggio macchina, attraverso l'assemblatore.

Spesso fanno eccezione alcuni compilatori che trasformano direttamente il linguaggio ad alto livello in codice macchina.

1.2.3 Periferiche di IO

Il **mouse** è stato inventato nel 1967 da *Doug Engelbart* nei laboratori della *Xerox*. Attualmente sfrutta la tecnologia ottica: attraverso le variazioni di luce provocate da alcuni led che illuminano il piano, il mouse è in grado di rilevare gli spostamenti.

Gli schermi **LCD** (*Liquid Crystal Display*) sono costituiti da alcuni cristalli che galleggiano in un fluido: ciascuno di essi corrisponde ad un pixel. Attraverso un campo elettrico è possibile ruotare di 90 gradi ogni singolo cristallo, che di conseguenza può impedire o meno il passaggio della luce secondo il fenomeno fisico della *luce polarizzata*.
L'immagine dunque viene rappresentata da una matrice di pixel: ciascuno di essi ha associata una componente rosso, verde, blu (sistema RGB). Questa immagine viene memorizzata in un *frame buffer*, una RAM che viene aggiornata fino a 100 volte al secondo.

1.2.4 Misurazione delle prestazioni

I moderni processori sono costruiti usando un segnale periodico che ne sincronizza le operazioni. Il *ciclo di clock* è l'intervallo di tempo che intercorre tra due colpi di clock. La frequenza è definita come $\frac{1}{\text{ciclo clock}}$. Il ciclo di clock è misurato in secondi, la frequenza in Hertz.

Il tempo necessario per l'esecuzione di un programma dipende da tre fattori:

- il numero di istruzioni dell'algoritmo.
- i cicli di clock per istruzione (CPI).
- la frequenza di clock.

$$\text{Tempo CPU} = \text{Num Istruzioni} \cdot \text{CPI} \cdot \text{Periodo Clock} = \frac{\text{Num Istruzioni} \cdot \text{CPI}}{\text{Frequenza Clock}}$$

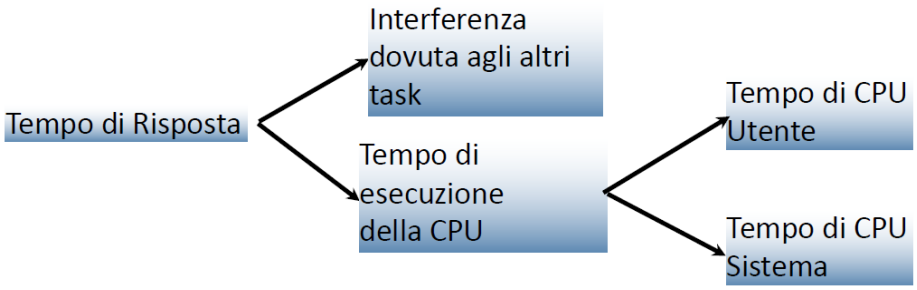


Figura 1.1: Diagramma del tempo di risposta

Un algoritmo è efficiente se viene strutturato in modo da risparmiare istruzioni e, per una particolare architettura, se utilizza le istruzioni più efficienti (quelle con un basso CPI, poichè il CPI dei diversi tipi di istruzione varia in base all'architettura che deve svolgere le istruzioni).
Il linguaggio di programmazione influenza il numero di istruzioni e il CPI: più di alto livello sono i costrutti, più lunghe sono le sequenze di istruzioni macchina ottenute traducendo il codice di partenza.
Il compilatore sicuramente influenza sia il numero di istruzioni che il CPI in base alla propria efficienza e ottimizzazione.
Anche l'*ISA* ha impatto sul numero di istruzioni, sul CPI, e sulla frequenza di clock attraverso la sua progettazione: essa può fornire istruzioni di basso o alto livello (più o meno istruzioni per eseguire un'operazione).

1.2.5 La legge di Moore

Negli scorsi anni le prestazioni dei calcolatori sono aumentati costantemente, secondo l'andamento previsto da *Moore* attraverso una legge informale.
Di recente si è assistito a una diminuzione dell'incremento tra una generazione e l'altra, sintomo di una evidente saturazione, casuata da limiti fisici della materia.

1.2.6 La barriera dell'energia

Dagli anni Ottanta ad oggi, la frequenza media dei processori è stata aumentata di più di mille volte, con conseguente aumento dei consumi di 30 volte.

$$\text{Potenza} = \text{capacità} \cdot \text{tensione}^2 \cdot \text{frequenza di commutazione}$$

Dove la frequenza di commutazione è legata alla frequenza di clock.
Incrementare la frequenza senza eccedere in consumi è stato permesso grazie un abbassamento della tensione di alimentazione (termine quadratico, quindi il più influente) da 5 V a 1.2 V. Al di sotto di questo valore avvengono fenomeni di tipo elettrostatico che portano il sistema in una condizione anomala.
C'è inoltre un limite alla capacità di estrarre la potenza prodotta dai processori (e il conseguente calore) tramite ventole o radiatori. Il processo di refrigerazione diventa molto costoso e difficilmente attuabile in dispositivi desktop e laptop.

La soluzione principale che è stata adottata consiste nelle *architetture multicore* sfruttando il concetto di *parallelismo*. Ciò comporta una maggiore difficoltà e responsabilità nella scrittura di codice da parte del programmatore (debugging complicato, bilanciare il carico di lavoro fra le CPU).

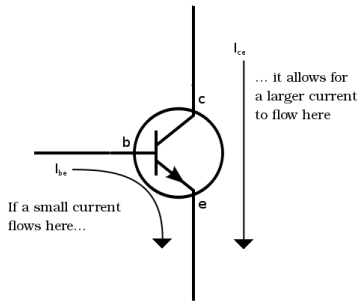
Capitolo 2

Aritmetica dei calcolatori

2.1 Informazione nei computer

2.1.1 I transistor

Tutti i computer moderni sono composti da *transistor* (detti anche triodi) che sono strutture bistabili, ovvero che possono assumere due stati come un interruttore: 0=spento e 1=acceso. Molti *transistor* collegati tra loro in una sorta di matrice possono rappresentare delle serie di porte logiche, ed è questa la struttura che sta alla base dell'architettura dei moderni calcolatori.



Un computer memorizza (e manipola) solo sequenze di 0 e 1 (sequenze di bit); anche l'ENIAC funzionava allo stesso modo, solo che al posto dei *transistor* era composto da *valvole termoioniche*, che in ultima analisi non erano differenti nel funzionamento (assumono sempre i due stati descritti in precedenza, solo più lentamente).

2.1.2 Interpretazione delle informazioni

In seguito le sequenze di bit che i computer elaborano/memorizzano possono essere interpretate in tantissimi modi diversi, tra cui numeri, caratteri, suoni, immagini, istruzioni e molti altri.
Alla base di tutte le interpretazioni che si possono dare alle stringhe di simboli (che nel caso del computer sono proprio i due stati del bit) sta il concetto di **codifica**. La **codifica** è appunto uno schema, una legge, un *mapping* che permette di tradurre prima ed interpretare poi stringhe di simboli. Nell'am-

bito dell'IT le codifiche sono fortemente caratterizzate dalla lunghezza (dal numero di bit) delle "parole" elementari della codifica. Ad esempio per rappresentare tutti i caratteri (tabella ASCII) utilizziamo 8 bit [256 caratteri]. Si osservi che essendo i *transistor* sistemi bistabili la base 2 (con cifre 0 e 1) è perfetta per rappresentare la codifica dei dati memorizzati/elaborati da essi.

2.2 I numeri

Limitiamoci al caso dei numeri per spiegare il concetto di **codifica**:
Ricordiamo che i metodi di rappresentazione numerica che studiamo sono posizionali, ovvero il peso di ogni cifra varia in base alla posizione che essa occupa.
In generale se una base è composta da B elementi essa ha B cifre (da 0 a B-1) utilizzate per scrivere ogni numero. Questa formula ci restituisce il valore di ogni numero scritto in una qualsiasi base B:

$$c_i c_{i-1} c_{i-2} ... c_0 = c_i \cdot B^i + ... + c_0 \cdot B^0$$

dove c_i è la cifra in posizione i.

2.2.1 Regole di conversione tra basi

(un elenco delle principali basi è troppo mainstream?)

Vediamo ora come operare la conversione tra le principali basi:

- **da base 2 a base 16**: partendo da destra, si dividono le cifre binarie in gruppi da 4 cifre ciascuno, e ognuno di questi corrisponderà a una cifra esadecimale; qualora il numero di cifre binarie non sia divisibile per 4, si completi con degli 0 in modo da ottenere gruppi da 4 (*esempio:* $(11011)_2$ *diventerà* $(0001\ 1011)_2$, *ovvero* $(1B)_{16}$);
- **da base 2 a base 8**: analogamente, si dividano le cifre binarie in gruppi da 3 cifre ciascuno, e ognuno di questi corrisponderà a una cifra ottale;
- **da una qualsiasi base B a base 10**: come visto sopra, si moltiplica ogni cifra c_i per B^i, dipendentemente dalla sua posizione;
- **da base 10 a una qualsiasi base B**:
  1. si consideri un numero x_{10} e una base B;
  2. si divida x per B;
  3. il resto della divisione è la cifra da inserire a sinistra nel numero convertito;

4. si assegni a x il quoziente della divisione;
5. si torni al punto 2 e si ripeta finché $x \neq 0$.

$1000_{10} = ?_2$

X	$X/2$	$X\%2$
1000	500	0
500	250	0
250	125	0
125	62	1
62	31	0
31	15	1
15	7	1
7	3	1
3	1	1
1	0	1

$1000_{10} = 1111101000_2$

$1000_{10} = ?_{16}$

X	$X/16$	$X\%16$
1000	62	8
62	3	14 (E)
3	0	3

$1000_{10} = 3E8_{16}$

Figura 2.1: Alcuni esempi di conversione

2.2.2 I naturali

Nella codifica binaria un numero naturale è rappresentato su k cifre binarie, dove con k cifre si possono rappresentare i numeri tra 0 e $2^k - 1$.
La codifica più comune per gli interi spesso usa i byte, sequenze di 8 bit, che tuttavia richiedono molte cifre per rappresentare un numero e quindi spesso, per semplificare la lettura, si usa scrivere i numeri in esadecimale (quindi scrivendo un quarto delle cifre rispetto alla binaria).

Somma e sottrazione Le operazioni somma e sottrazione funzionano allo stesso modo del sistema utilizzato nella numerazione decimale con il riporto. Si guardino gli esempi qui sotto riportati, già di per sè esplicativi del processo:

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

Figura 2.2: Esempio di somma

1	1	0	1	0	1	1	0	-
	1	1	1	0	1	0	1	=
	1	1	0	0	0	0	1	

Figura 2.3: Esempio di differenza

Moltiplicazione La moltiplicazione in base binaria si può semplificare con il metodo dello shifting, che è più facile illustrare con un esempio:

				1	1	0	1	x
			1	0	0	0	1	=
				1	1	0	1	+
						-		+
					-			+
				-				+
1	1	0	1					
1	1	0	1	1	1	0	1	

Figura 2.4: Esempio di prodotto

Si noti che quando è presente un 1 si riscrive il numero (nella posizione corrispondente) mentre quando è presente uno 0 semplicemente non si scrive nulla (e si va solo avanti con le posizioni).

2.2.3 Gli interi

Finora abbiamo visto come si codificano solo i numeri naturali, ma esistono metodi di codifica che permettono di rappresentare anche i numeri negativi (quindi l'insieme degli interi). Le tecniche più comuni sono: modulo e segno, complemento a 1 (*CA1*) e complemento a 2 (*CA2*).

Codifica con Modulo e Segno: Idea semplice: si usano $k-1$ bit per rappresentare il valore assoluto (modulo) del numero ed un bit per codificare il segno (0=positivo, 1=negativo). In questo modo con k bit si possono codificare valori tra $-2^{k-1}+1$ e $+2^{k-1}-1$.
N.B.: esistono due codifiche per lo 0, -0 e $+0$, e questo è un spreco.

Codifica in Complemento a 1: L'idea alla base è semplice, il primo bit indica sempre il segno e per ottenere l'opposto di un numero positivo si invertono tutti gli 0 in 1 e gli 1 in 0 (e lo stesso si fa per ottenere l'opposto di un numero negativo). Si hanno ancora due rappresentazioni per +0 e -0 (rispettivamente, utilizzando 4 bit, 0000 e 1111). È più facile da sommare rispetto al modulo e segno ma solo se il bit significativo non da riporto. Quando si sommano due numeri in *CA1*:

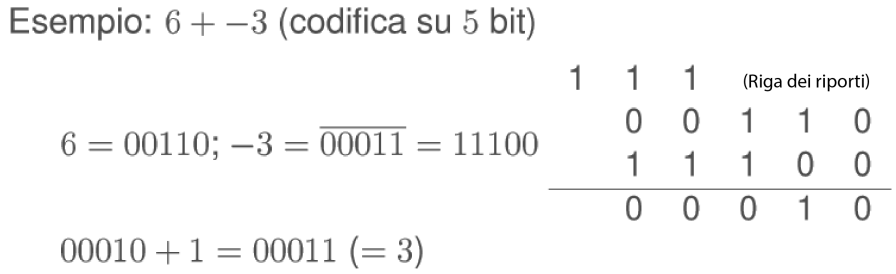


Figura 2.5: Esempio di somma in CA1

- Si deve verificare che i riporti delle prime due cifre significative siano uguali, altrimenti il numero che si ottiene non è rappresentabile in CA1 su k bit.
- Alla fine si deve sommare il riporto della prima cifra al risultato ottenuto (ultima riga).

Codifica in Complemento a 2: Per tradurre un numero da intero con segno a complemento a 2 (*CA2*) basta invertire tutti i bit e poi sommare 1. Viceversa, per convertire da *CA2* a binario, si sottrae 1 e poi si invertono tutti i bit.
Ancora una volta il bit più significativo indica il segno, ma in questo caso la codifica dello 0 è unica, quindi con k bit si possono rappresentare i numeri da -2^{k-1} a $2^{k-1} - 1$. Anche la somma è più semplice ma questo lo vedremo in futuro.

Somma algebrica in Complemento a 2

L'utilizzo della codifica *CA2* permette di eseguire somme algebriche senza alcuna differenza operativa, ma rende necessario prestare attenzione agli *overflows*.
Con il termine *overflow* si intende la situazione in cui, date in input delle informazioni codificate in un numero k di bit arbitrariamente scelto, lo stesso

numero k di bit non è sufficiente a codificare le informazioni in output.
Diamo subito un esempio: supponiamo di voler eseguire $-64-8$ utilizzando il *CA2* e $k = 7$ bit. Ricordiamo che:

$-64 = 1000000_2 = \overline{1000000} + 1 = 0111111 + 1 = 1000000$ in *CA2;*
$-8 = 0001000_2 = \overline{0001000} + 1 = 1110111 + 1 = 1111000$ in *CA2*.

A questo punto eseguiamo la somma algebrica

$$\begin{array}{cccccccc} (1) & & & & & & & \\ & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Figura 2.6: (Il numero fra parentesi indica l'ultimo riporto, che chiaramente non possiamo inserire poiché sforerebbe i 7 bit)

Osserviamo così che, per $k = 7$ bit, $-64 - 8 = 0111000_2 = 56_{10}$, che ovviamente non ci piace; concludiamo quindi che la somma algebrica $-64-8 = -72$ *non* è rappresentabile con soli $k = 7$ bit.

Possiamo dire che, in generale, dati due numeri a, b aventi lo stesso segno, avremo problemi di *overflow* se

- $a + b > 2^{k-1} - 1$
- $a + b < -2^{k-1}$

ricordando che $-2^{k-1} \leq a, b \leq 2^{k-1} - 1$.

Aritmetica modulare

Incosapevolmente noi utilizziamo l'*aritmetica modulare* quasi continuamente, nella nostra quotidianità; l'esempio più comune è quando sommiamo le ore, che sono organizzate in modulo 24 (da cui la denominazione ufficiosa di "Matematica dell'orologio"), ma anche minuti e secondi (modulo 60) e gli angoli (modulo 360).
Diamone una descrizione intuitiva: fissato un numero n finito di valori, arrivato al numero più grande $(n-1)$, al successivo avrò che $(n-1)+1 = 0$.
Per completezza, forniamo ora la relazione di equivalenza di due numeri in modulo n:

$$a \equiv b \longleftrightarrow a\%n = b\%n$$

Quando noi eseguiamo delle operazioni sui numeri in base 2 su k bit, stiamo lavorando in un ambiente in modulo 2^k, e calcolando $a + b$ stiamo in realtà trovando $(a + b)\%2^k$, e i due valori saranno uguali se e solo se $a + b < 2^k$, perché altrimenti ogni valore successivo ricomincerebbe da 0: questo è l'*overflow*.
In particolare quando lavoriamo su numeri interi in *CA2* dobbiamo ricordare che, se $-2^{k-1} \leq a + b \leq 2^{k-1} - 1$ non avremo *overflow*, altrimenti

- se $a + b < 2^{k-1} - 1$, otterremo $a + b + 2^k \longrightarrow$ *OVERFLO(W)!*
- se $a + b > -2^{k-1}$, otterremo $a + b - 2^k \longrightarrow$ *OVERFLO(W)!*

2.2.4 I Reali

Non è sempre semplice codificare un numero reale $\alpha \in \mathbb{R}$, poiché di questo insieme fanno parte i numeri cosiddetti *irrazionali*, che presentano infinite cifre non periodiche dopo la virgola (si pensi ad esempio a $\pi = 3,14159...$, o $e = 2.71828...$, o ancora $\sqrt{2} = 1.41421...$).
Alcuni numeri non possono quindi essere rappresentati con un numero finito di bit, per cui ne rappresenteremo solo un'approssimazione, e per farlo abbiamo sviluppato due metodi: la rappresentazione a **virgola fissa** (o *fixed point*) e quella a **virgola mobile** (o *floating point*).

Codifica in virgola fissa

Questo sistema di cofidica prevede che, posto un numero k di bit per la rappresentazione del numero, si impieghino $k - f$ bit per rappresentare la parte intera, e i restanti f bit per rappresentare la parte razioneale:

$$\overbrace{\underbrace{c_{k-1}c_{k-2}\cdots c_f}_{k-f}\underbrace{\cdot c_{f-1}\cdots c_0}_{f}}^{k}$$

Possiamo scrivere una formula generale più compatta. Supponiamo di voler convertire un numero x_{10}:

$$x_{10} = \sum_{i=0}^{k-1} c_i B^{i-f} = \sum_{i=0}^{k-1} c_i B^i \cdot B^{-f} = (\sum_{i=0}^{k-1} c_i B^i) \cdot B^{-f}$$

Detta in altri termini, convertiamo il numero da base 10 come se non avesse la virgola e poi moltiplichiamo per B^{-f}.
Presentiamo vantaggi e svantaggi di questo tipo di codifica:

- le operazioni si possono svolgere senza alcuna differenza, se non l'accortezza di scalare il risultato;

- il peso per la CPU è ridotto;
- non ci sono errori di approssimazione, *se il valore è rappresentabile*;
- iniziamo ad avere problemi quando lavoriamo con un ampio spettro di ordini di grandezza.

(esempio di conversione?)

Codifica in virgola mobile

Un qualsiasi numero reale può essere scritto anche nella forma $x = M \cdot B^E$ dove M si dice *mantissa*, E *esponente* e B è la base della rappresentazione (che nel nostro caso è la leggendaria base 2).
Questo metodo di rappresentazione, del tutto analogo notazione scientifica standard, è implementato anche nell'informatica in più modi diversi ma alla base di tutti si riconosce lo stesso schema:
x può essere rappresentato su k bit utilizzando m bit per il campo mantissa M ed $e = k - m$ bit per il campo esponente E; quest'ultimo permette in un certo senso di "spostare" la virgola; con esponenti negativi si avranno x "piccoli" e viceversa con esponenti positivi si avranno x "grandi".
Lo standard più comunemente utilizzato è l'IEEE 754 che, nei numeri *float* a precisione singola (dove si usano 32 bit), ha questa struttura:

1 bit	8 bit	23 bit
Bit di segno	Esponente	Mantissa

Il bit di segno ha la funzione che già conosciamo, mentre il campo E può contenere valori compresi tra 0 e 255; questi ultimi due valori vengono utilizzati per funzioni speciali, mentre i valori da 1 a 254 compresi codificano l'esponente effettivo dei calcoli, che diremo exp[1], in questo modo:

$$exp = E - 2^{e-1} - 1$$

Nel nostro caso, $exp = E - 127$. L'ultimo campo, la *mantissa*, codifica un numero intero.

Come calcolare il valore di un *float* Il primo bit è utilizzato per indicare il segno, i successivi 8 per E e i restanti 23 per la mantissa. La decodifica in decimale assume due forme in base al valore di E:

- se $E = 0$, allora il numero in base 10 è $x_{10} = 0.M \cdot 2^0$;

[1]La notazione *exp* è stata introdotta da noi autori della dispensa perchè ci è sembrata più chiara ed agile di quella utilizzata dal prof., chiediamo venia se questo genererà confusione

- se $E > 0$, allora il numero in base 10 è $x_{10} = 1.M \cdot 2^{exp}$, dove exp è definito come qui sopra descritto.

Nel primo di questi due casi i numeri si dicono *denormalizzati*, essi rappresentano tutti i numeri compresi tra 0 e 1; nel secondo caso i numeri si dicono *normalizzati* [questa distinzione è stata aggiunta per migliorare la precisione della notazione con numeri molto vicini allo 0].
Per tradurre in decimale un numero in virgola mobile si deve quindi per prima cosa determinare il segno, poi si calcola l'esponente effettivo dei calcoli ($exp = E-127$) e da qui, simmetricamente a prima, si procede in due modi:

- se $exp \neq -127$ si parla di numero *normalizzato* e quindi si aggiunge un bit 1 alla mantissa in modo da ottenere il numero $1.M$ e si sposta la virgola di exp posizioni e per completare il tutto si traduce il numero binario così ottenuto in decimale;
- se $exp = -127$ si parla di numero *denormalizzato* e quindi si aggiunge un bit 0 alla mantissa in modo da ottenere $0.M$ e come prima si sposta la virgola di (127) posizioni e poi si decifra il numero (binario) così ottenuto;

Una volta stabilito quale delle precedenti due opzioni riguarda il nostro caso spostiamo la virgola di exp posizioni nel numero $1.M$ oppure $0.M$ (verso destra se $exp > 0$ e viceversa), decodifichiamo il numero ottenuto come se fosse un binario in virola fissa.

Tabella riassuntiva

Ecco una breve tabella riassuntiva dell'interpretazione delle stringhe nella rappresentazione IEEE 754:

Categoria	Esponente	Mantissa
Zeri	0	0
Num. denormalizzati	0	non zero
Num. normalizzati	1-254	qualunque
Infiniti	255	0
Nan (not a number)	255	non zero

Esempio di decodifica di un float

Dato il numero float (IEEE 754) 0 10000101 00110111000000000000000, traducilo in decimale.

Consideriamo i diversi campi:

MSB	Esponente	Mantissa
0	10000101	00110111000000000000000

1) Il numero è positivo (MSB=0)

2) L'esponente dei calcoli *exp* è *E - 127* = 10000101 - 01111111 = 133 - 127 = 6 (=> l'esponente è diverso da 0, si tratta di un numero normalizzato)

3) la mantissa va moltiplicata per $2^{exp} = 2^6$ e dato che il numero è normalizzato uso 1.M:

$1.M \times 2^6 = 1.00110111000000000000000 \times 2^6 = 1.00110111 \times 1000000 = 1001101.11$

4) una volta ottenuto il numero 1001101.11 lo si traduce (da virgola fissa a decimale)

$2^6 + 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 77{,}75$

Figura 2.7: Decodifica da IEEE 754 a decimale

Capitolo 3

Codifica del testo

3.1 La codifica ASCII

Oltre alle codifiche per numeri naturali, interi e reali viste nel capitolo precedente, sono state introdotte delle codifiche per rappresentare sequenze di caratteri. Una delle più note è la codifica **ASCII** (*American Standard Code for Information Interchange.*), che utilizza 7 bit per rappresentare ciascun carattere (quindi $2^7 = 128$ caratteri disponibili in totale).
Attraverso la codifica *ASCII* è possibile rappresentare tutte le lettere dell'alfabeto anglosassone (lettere maiuscole, minuscole, numeri, punteggiatura, ecc.).
Questa codifica presenta uno svantaggio: seppur utilizzi solamente 7 bit, un byte è formato da 8 bit. Ciò implica che il primo bit di ciascun carattere viene impostato a 0 per default.

3.2 La codifica ASCII estesa

A differenza dell'*ASCII* tradizionale, l'*extended ASCII* sfrutta anche l'ottavo bit per codificare caratteri addizionali (quindi $2^7 = 128$ caratteri disponibili in totale).
Non esiste un unico standard esteso: sono state implementate diverse versioni, ciascuna in grado di supportare diversi alfabeti specifici. In particolare, i byte con valore minore di 128 sono comuni a tutte le implementazioni della codifica ASCII estesa.

3.3 La codifica UNICODE

La necessità di avere una codifica univoca ha portato ad aumentare fino a 32 il numero di bit per la rappresentazione di un carattere attraverso la codifica **Unicode**. Anche qui non mancano differenti formati di codifica:

- *UTF-32*: ogni simbolo è composto da 32 bit.
- *UTF-16*: ogni simbolo è composto da 16 o più bit (lunghezza variabile).
- *UTF-8*: ogni simbolo è composto da 8 o più bit; mantiene la compatibilità con ASCII.

Input e output da file Come si è visto precedentemente, è dunque possibile codificare dati numerici in diverse modalità. Quando si scrive un programma che si interfaccia con file testuali, si consiglia caldamente di salvare i dati numerici in formato binario e non in forma di stringa. Questa accortezza permette di ridurre la dimensione in byte del file finale prodotto.

Capitolo 4

Le reti logiche

4.1 Introduzione

Come ben sappiamo, a differenza di *ENIAC*, i computer moderni sono realizzati da un'enorme quantità di circuiti elettronici. Trattandosi di eletronica digitale, a livello hardware è possibile lavorare con due livelli fondamentali:

- 1: alto, asserito; associato alla tensione di alimentazione *Vdd*.
- 0: basso, negati; associato alla massa (tensione vicina a 0).

Nel passaggio da un livello all'altro si presenta lo stato *transitorio*, in cui la tensione assume valori intermedi fra zero e Vdd.
In particolare, i **circuiti logici** (formati dalle **porte logiche**) vengono utilizzati per trasformare alcuni valori logici in ingresso in altri valori logici in uscita. I circuiti logici si differenziano principalmente in:

- *Combinatorie*: l'uscita dipende unicamente dal valore di ingresso (non sono dotati di memoria).
- *Sequenziali*: l'uscita dipende dal valore di ingresso e dalla storia degli ingressi precedenti (sono dotati di memoria, detta "stato" della rete).

È possibile riassumere il comportamento di una rete logica attraverso una *tabella di verità*, che descrive i diversi output al variare dei diversi input.

4.2 L'algebra di Boole

Esisotno principalmente tre operazioni di base:

- **AND**: $A \cdot B$; produce 1 se entrambi gli operandi sono 1, 0 altrimenti.
- **OR**: $A + B$; produce 0 se entrambi gli operandi sono 0, 1 altrimenti.
- **NOT**: $\overline{A}$; inverte il valore logico.

4.2.1 Regole di semplificazione

Di seguito vengono riportate alcune regole di semplificazione:

- Identità: $A + 0 = A$, $A \cdot 1 = A$
- Regola "zero e uno": $A + 1 = 1$, $A \cdot 0 = 0$
- Regola dell'inversa: $A + \overline{A} = 1$, $A \cdot \overline{A} = 0$
- Regola commutativa: $A + B = B + A$, $A \cdot B = B \cdot A$
- Regola associativa: $A + (B + C) = (A + B) + C$, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Regola distributiva: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, $A + (B \cdot C) = (A + B) \cdot (A + C)$
- De Morgan: $\overline{A \cdot B} = \overline{A} + \overline{B}$, $\overline{A + B} = \overline{A} \cdot \overline{B}$

Proprio grazie alla regola di De Morgan, si può affermare che la porta **NAND** (not AND) e la porta **NOR** (not OR) sono universali. Ciò significa che utilizzando unicamente porte NAND (rispettivamente NOR) è possibile costruire tutte le altre porte logiche.

4.2.2 La tavola di verità e i mintermini

Input			Output		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Tabella 4.1: Esempio di tabella di verità

Si può verificare che:

$$D = A + B + C$$
$$F = A \cdot B \cdot C$$
$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

In particolare, l'ultima espressione è stata ottenuta come somme di prodotti attraverso i cosiddetti *mintermini*. Per ciscun 1 presente nella colonna E, si scrive il prodotto degli input (asseriti se pari a 1, negati se pari a 0); il risultato dell'espressione è pari alla somma di tali prodotti.

4.3 Le porte logiche

A seguire vengono rappresentate, attraverso i corrispettivi simboli, le porte logiche implementate attraverso gli operatori booleani fondamentali:

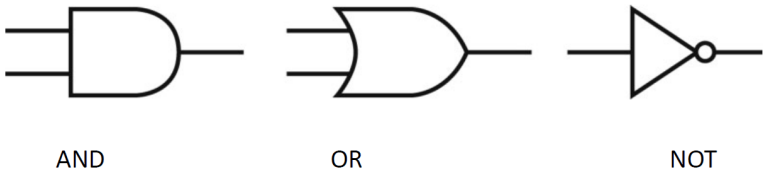
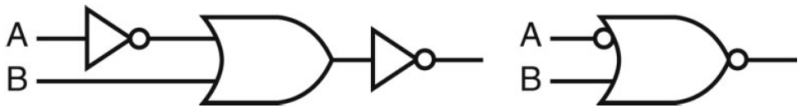


Figura 4.1: Rappresentazione di AND, OR e NOT

Si ricorda inoltre che è possibile combinare fra loro diverse porte logiche:



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

Figura 4.2: Combinazione di porte logiche

4.4 Alcuni circuiti degni di nota

4.4.1 Decoder

L'ingresso del *decoder* svolge il ruolo di selettore: ricevendo in input il valore n, si accenderà l'n-esima uscita. A seguire l'implementazione di un decoder a 3 bit:

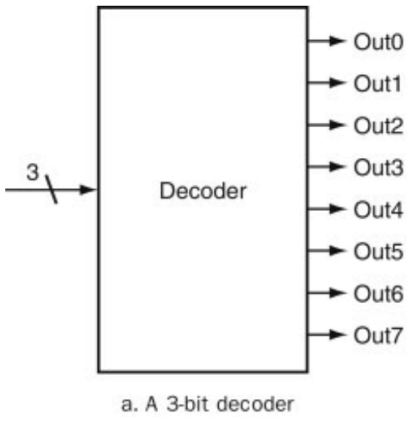


Figura 4.3: Decoder a 3 bit

Input			Output							
In2	In1	In0	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Tabella 4.2: Tabella di verità per un decoder a 3 bit

4.4.2 Multiplexer

Prevede un input con valore S utilizzato come selettore. L'output del *multiplexer* è il valore dell'input S-esimo.

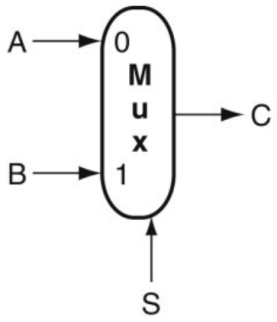


Figura 4.4: Multiplexer a 1 bit

È possibile costruire un multiplexer attraverso un decoder:

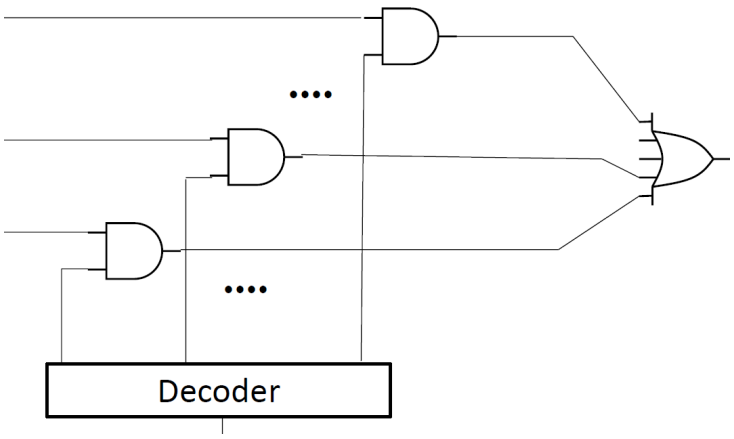


Figura 4.5: Multiplexer a n vie

Molto spesso, per operare con dati complessi, si compongono degli array di elementi elementari. Segue un esempio di come sia possibile costruire un multiplexer con *bus* a 32 bit utilizzando un array di multiplexer a 1 bit.

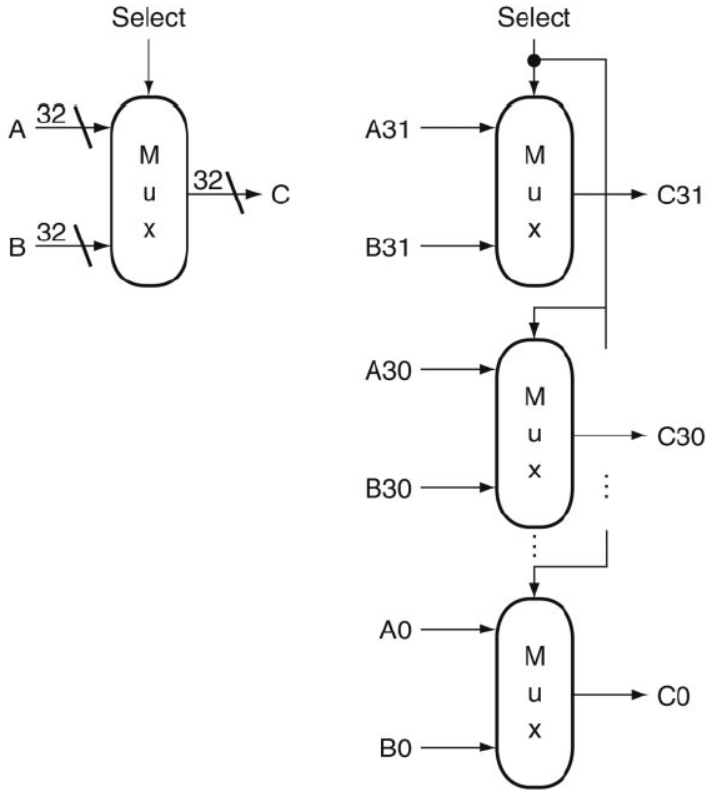


Figura 4.6: Multiplexer a 32 bit

4.4.3 Programmable Logic Array (PLA)

Il *programmable logic array* si compone di due strutture: una barriera di AND e da una barriera di OR. La dimensione totale del *PLA* è data dalla somma del piano AND (numero di mintermini) e del piano OR (numero di uscite).
Inoltre, il PLA implementa porte logiche solamente per le configurazioni che producono 1 in uscita. In aggiunta, se un mintermine è condiviso tra varie uscite, lo si può riutilizzare. Successivamente seguono due diverse implementazioni della rete logica descritta nel paragrafo 4.2.2.

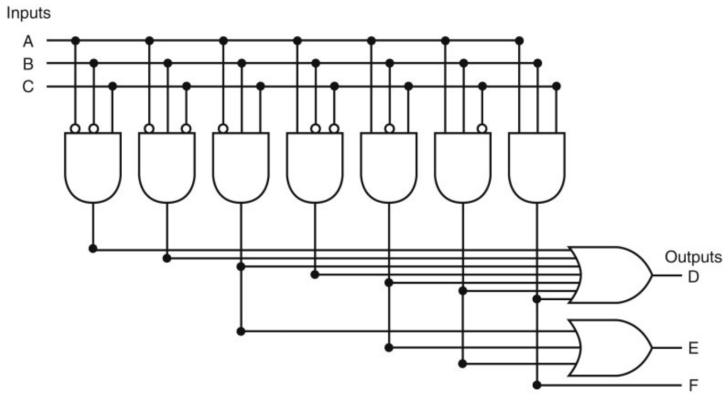


Figura 4.7: Esempio di rete logica

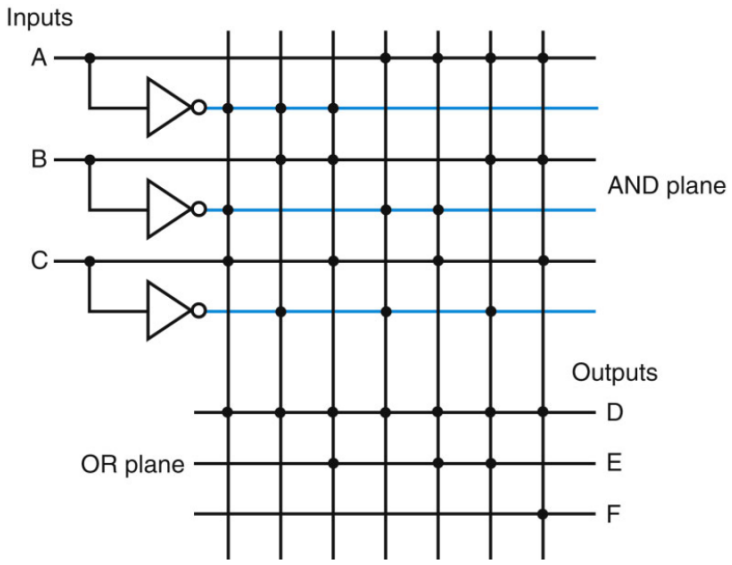


Figura 4.8: Esempio di rete logica implementata attraverso PLA

4.5 Il concetto di *costo*

Come appena visto, le reti logiche possono essere implementate in maniera differente. Si definisce *costo* di una rete logica la somma del numero di porte e del numero di ingressi della rete. Quando si analizza una rete, è importante saperne trovare l'implementazione con costo minimo. Principalmente si può procedere in due modalità:

- regole di semplificazione matematiche dovute all'algebra di Boole (vedi paragrafo 4.2.1).

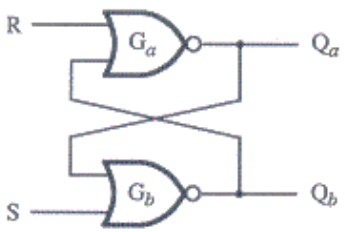
- metodi basati su rappresentazioni grafiche (mappe di *Karnaugh*), nei casi più semplici.

4.6 Le reti sequenziali

Come si è visto, le reti combinatorie non hanno memoria degli stati del passato: in ogni istante di tempo l'uscita dipende solamente dagli ingressi nell'istante considerato. Tuttavia, in certe applicazioni è necessario introdurre una vera e propria memoria nel sistema.
La memoria in una rete logica si ottiene attraverso una *retroazione*, che consiste nel ridirezionare alcune porte di uscita in ingresso ad altre porte del medesimo circuito, in maniera tale da formare un anello.
Seppur questo meccanismo offra delle potenzialità enormi, l'analisi e la sintesi di una rete logica è molto più complessa rispetto a quella delle reti combinatorie.

4.6.1 Latch S-R

A seguire un'implementazione di una rete sequenziale *latch S-R* (*set-reset*) costruita attraverso due porte NOR:



(a) Network

S	R	Q_a	Q_b	
0	0	0/1	1/0	(No change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Truth table

Figura 4.9: Esempio di latch a porte NOR

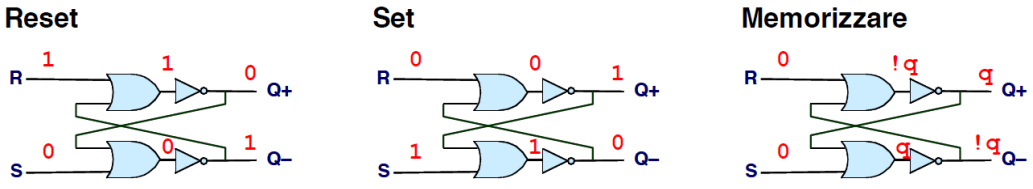


Figura 4.10: Operazioni di un latch a porte NOR

Ma, considerando che i segnali non si propagano in tempo nullo, ciò potrebbe causare dei problemi circa la correttezza del segnale stesso. La soluzione è quella di aggiungere un ulteriore segnale, in grado di temporizzare il circuito: il *clock*. Un latch dotato di temporizzazione viene chiamato *gated latch*.

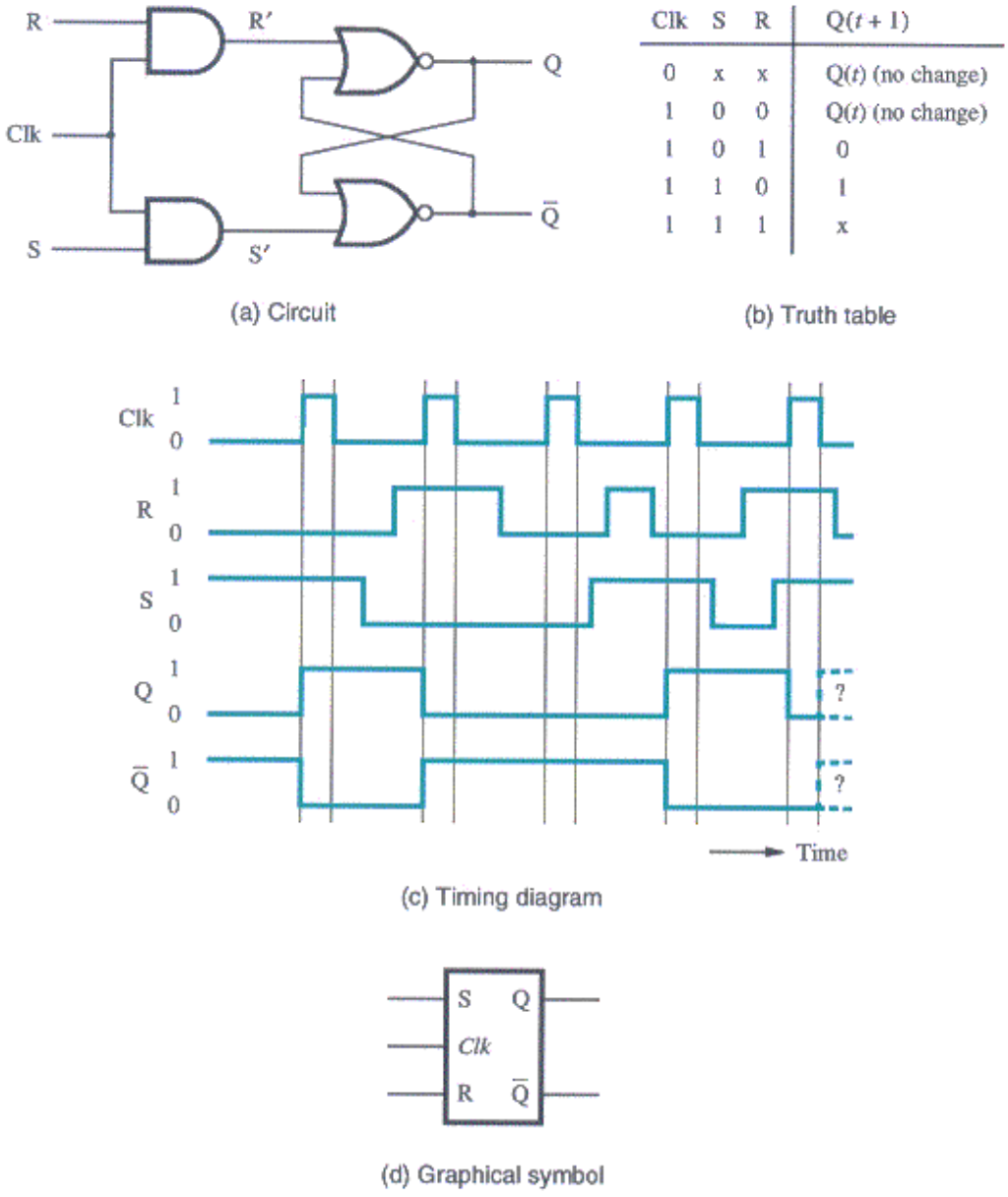


Figura 4.11: Esempio di gated latch

Il difetto principale di un latch R-S è quello di avere uno stato indecidibile: l'uscita non può essere nota con certezza quando entrambi gli ingressi sono a 1.

Una soluzione è implementata attraverso *latch* tipo *D*: gli ingressi al circuito sono ottenuti da un'unica variabile (di cui se ne fa il negato). Altrimenti, attraverso circuiti *flip-flop master-slave* è possibile ottenere che l'uscita del circuito commuti esattamente al termine dell'impulso di clock.
In particolare, i latch di tipo D vengono utilizzati per implementare i registri della CPU.

Capitolo 5

Introduzione al linguaggio Assembly

5.1 Perchè esiste Assembly

Le CPU capiscono esclusivamente le istruzioni che sono scritte nel loro specifico linguaggio macchina, che corrisponde ad una sequenza di bit 0 e 1. Al giorno d'oggi programmiamo praticamente solo ad alto livello, poiché programmare in linguaggio macchina sarebbe molto più complesso e dispendioso; il mezzo che ci permette di intermediare tra l'alto livello a cui programmiamo e la macchina è l'Assembly.
Il vantaggio di Assembly è l'utilizzo di codici mnemonici (le keyword del linguaggio) che al programmatore risultano più semplici delle stringhe di 0 e 1. Si deve però notare che Assembly, essendo di così basso livello, è un linguaggio molto specifico, che si adatta solo ad una certa struttura (anche se esistono architetture che possono essere gestite da più versioni di Assembly), ad esempio un linguaggio Assembly per ARM non può essere usato su architetture Intel.

5.2 Come funziona Assembly

5.2.1 L'Assembler

Una volta che il programmatore ha scritto il suo algoritmo e questo è stato codificato in Assembly dal calcolatore, chi si occupa della traduzione dell'Assembly?
L'Assembler, che è appunto il compilatore dell'Assembly (es.: Se ho un'istruzione del tipo ADD A B l'Assembler la trasforma in codice binario).

L'insieme di tutte le istruzioni specifiche conosciute da un'architettura è detto **ISA** (Instruction Set Architecture);

5.2.2 Instruction Set Architecture

l'ISA è un insieme di regole e procedure che specifica come ogni istruzione deve essere strutturata, codificata ed anche eseguita; in pratica esso specifica sia la semantica che la sintassi, che cosa si deve fare per compiere delle istruzioni e come si fa.
L'ISA specifica anche di che tipo sono i registri (general-purpose o specializzati), come accedere ad essi (molto vicini alla CPU contengono i dati che devono essere utilizzati in maniera super-rapida) ed alla memoria (indirizzamento).
Per comprendere bene le istruzioni che componfigono l'ISA conviene ripassare le fasi di funzionamento di una CPU.

5.2.3 Funzionamento di una CPU

Il funzionamento di una CPU segue il seguente schema:

- **Fetch**: In questa fase viene letta l'istruzione da eseguire
  1. l'istruzione da eseguire, puntata dal Program Counter (registro che tiene conto della successiva istruzione da eseguire), viene copiata nell'Address Register (indirizzo di accesso al bus di sistema);
  2. i dati dell'istruzione vengono quindi dalla RAM al Data Register (che tiene conto dei dati da leggere o scrivere);
  3. i dati vengono salvati in un registro invisibile al programmatore;
  4. viene incrementato il program counter;
- **Decode**: L'istruzione qui dev'essere decodificata (spacchettata in una serie di campi) che contengono le varie caratteristiche dell'istruzione
- **Execute**: alla fine l'istruzione viene eseguita, essa può essere aritmetica, logica, di gestione della memoria, ecc.

Ricapitolando il ciclo dell'esecuzione delle istruzioni segue questo schema: **fetch**>**decode**>**execute**.
Queste operazioni vengono scandite dal clock, lo svolgimento delle istruzioni è prevalentemente lineare tuttavia esistono delle istruzioni che possono modificare il *Program Counter* che contiene l'indirizzo della prossima istruzione; istruzioni simili sono i cicli, i salti o le selezioni.

5.2.4 Ritornando ad Assembly

Nell'Assembly di fatto non esistono istruzioni come i cicli, però queste vengono implementate attraverso istruzioni che modificano il Program Counter. Complessivamente un programma in Assembly può essere descritto come una lista di istruzioni di tre tipi:

- Operazioni aritmetiche/logiche (o anche algebriche e booleane)
- Operazioni di gestione dati (istruzioni di indirizzamento)
- Operazioni di controllo del flusso del programma (operazioni sul Program Counter)

Generalmente possiamo utilizzare le istruzioni in due modi diversi, o inserendo direttamente i dati nel corpo dell'istruzione, ma questa è una grande limitazione poiché le istruzioni hanno dimensioni preimpostate (o comunque limitate), o inserendo nelle istruzioni gli indirizzi di alcuni registri (dove posso stoccare i dati che necessito); esiste un'ultima possibilità, che ci permette di memorizzare nelle istruzioni gli indirizzi dei dati che stocchiamo in memoria (così da poter sfruttare le quantità enormi di memoria di un calcolatore) e poi prelevare i dati necessari dalla memoria e caricarli nei registri (così da avere la velocità di accesso dei registri).
Si deve tenere in conto che però i registri sono limitati (in alcune architetture ARM sono addirittura solo 4) e devono anche essere ripuliti dopo ogni utilizzo (a questo pensa la CPU).

Registri Esistono diversi tipi di registro, sia generali che specifici; un esempio di registro specifico è il *registro zero*, che contiene tutti zeri e viene utilizzato per semplificare e diminuire le istruzioni dell'ISA. Per copiare un registro in un altro, invece che implementare una funzione apposta, si somma il registro con il *registro zero* e si salva il risultato in un terzo registro (che sarà la copia del primo).

5.2.5 Diversi tipi di ISA

Nel tempo si sono sviluppati due diversi approcci verso gli ISA, il *Complex Instruction Set Computer* (**CISC**) ed il *Reduced Instruction Set Computer* (**RISC**) che appunto differiscono in questo:

- **CISC**: comprende molte istruzioni, anche complesse, e quindi ha molto meno bisogno di registri e la scrittura di programmi Assembly risulta più facile; tuttavia per implementare le sue istruzioni (come già detto complesse) aumenta la difficoltà nella progettazione di l'hardware adatti.

- **RISC**: comprende solo istruzioni di base o comunque poche istruzioni e relativamente semplici, in questo modo semplifica l'implementazione della CPU e della sua architettura, ma ha bisogno di molti più registri per caricare le istruzioni temporanee (le intermedie necessarie per compiere istruzioni più complesse) ed in più rende più difficile la programmazione in Assembly (il suo linguaggio conosce solo le istruzioni basilari, quindi quelle complesse dovranno essere descritte dal programmatore).

5.2.6 I tipi di istruzioni

Istruzioni algoritmiche/logiche Le istruzioni aritmetiche e logiche spettano alla ALU che è il "chip" che svolge effettivamente (che esegue) le istruzioni;

Istruzioni di gestione dati Le istruzioni di gestione (o movimento) dati sono svolte da altri elementi specifici adibiti proprio allo spostamento di dati tra registri, da registri a memoria e viceversa.
Quando la CPU esegue un programma ha bisogno di due elementi: operandi e destinazione (dove salvare il risultato). Le due tipologie di ISA offrono alternative diverse rispetto alla posizione dove mettere questi elementi per poi renderli utilizzabili dalla CPU: l'ISA RISC, per semplicità e velocità, salva tutto nei registri, l'ISA CISC riesce ad utilizzare/mettere questi elementi anche nella memoria (rendendo anche la programmazione in Assembly più semplice).

Istruzioni condizionali/di controllo del flusso Per esprimere questo tipo di istruzioni, alcune ISA *saltano* nel codice (modificando il *Program Counter*) sfruttano il contenuto di registri generici; ad esempio, eseguono l'istruzione successiva solo se un certo registro è uguale ad un altro (questo richiede l'implementazione del confronto tra registri però, che è una classica operazione complessa da CISC); altri ISA controllano se sono presenti degli elementi detti falg contenuti in un apposito registro (*registro flag*) che segnalano determinate condizioni.
Le istruzioni di controllo di flusso sono svolte da alcuni elementi che gestiscono il *Program Counter*: quando non ci sono *salti* incrementano il PC di 4 in 4 (o 8 in 8 a seconda delll'architettura), quando si verificano *salti* invece le reti logiche hardware che gestiscono il PC devono cambiarlo ma tenere in memoria il suo indirizzo effettivo, per riprendere da lì al termine della subroutine.

Curiosità: nell'architettura ARM tutte le istruzioni sono condizionali, mentre altre architetture hanno solo poche istruzioni dedicate al salto condizionale.

5.2.7 Confronto tra CISC e RISC

Una volta osservato tutto ciò potrebbe venire spontaneo chiedersi quale delle due ISA sia la migliore, ma la risposta dipende dal contesto; le due modalità si sono sviluppate per essere ottimali in ambiti diversi (con obiettivi differenti).
Esistono tuttavia ISA ibride, che derivano da entrambe le architetture, come ad esempio l'ARM, che è presente sulla maggior parte degli smartphone odierni. Questi ibridi sono detti RISC "*pragmatici*" e combinano parte della flessibilità e complessità CISC con parte della semplificazione e regolarità RISC.

5.2.8 Accesso alla memoria

Per svolgere questa funzione in RISC vi sono solo le istruzioni *load* e *store*, mentre per CISC sono presenti istruzioni più generiche; in ogni caso l'argomento dell'istruzione è sempre *<memory location>*.
Ci sono varie modalità di indirizzamento che propongono che in questo campo siano scritti indirizzi diversi:

- **Indirizzamento assoluto**: nel campo (operando dell'istruzione) indico l'indirizzo di memoria (difetto: posso indicare pochi indirizzi poiché ho pochi bit);
- **Indirizzamento indiretto**: l'indirizzo di memoria contenuto in un registro (e nell'argomento *<memory location>* ho l'indirizzo del registro che contiene);
- **Base + spiazzamento**: l'indirizzo è ottenuto shiftando (o manipolando) il contenuto di un registro: ho un registro base con un indirizzo base ed un secondo registro che shifto (a partire dal base), come argomento <memory location> scrivo di quanto devo shiftare questo secondo registro
- **Combinazioni** varie delle modalità precedenti, che permettono di accedere a locazioni di memoria in maniera molto fine (es.: base + indice e base + indice + spiazzamento);

Se si vuole questo può essere osservato come un esempio di confronto tra i due tipi di ISA, nell' ISA CISC si può svolgere "*base + spiazzamento*" direttamente in una stessa istruzione (espressione), mentre nell'ISA RISC si necessita di più istruzioni.

5.2.9 Application Binary Interface

L'ISA definisce numero, nome e funzione dei registri (*general-purpose* o specializzati), ma questo non basta, perché non abbiamo risposte a domande come: quali registri posso usare? Quali posso modificare durante una *subroutine*? Come passo i valori dei parametri e di ritorno?
Per questo c'è il bisogno dell'*Application Binary Interface* (ABI), che fornisce una sorta di protocollo dell'utilizzo dei registri: in pratica è proprio una raccolta di regole sull'utilizzo dei registri.
Un programma è composto da tanti metodi e funzioni, e questi vengono mappati in *subroutine* (un pezzo monolitico di istruzioni) dal linguaggio Assembly. Per esempio, quando lancio l'istruzione $r = funz(arg1, arg2, arg3, \dots)$; $funz$ questa è una *subroutine* e devo decidere quali registri posso utilizzare per immagazzinare gli input della funzione, dove posso salvare il suo output, quali registri essa può modificare liberamente. Le risposte a queste questioni vengono risolte dall'ABI che definisce a livello software le **convenzioni di chiamata** che specificano come utilizzare i registri all'invocazione di una *subroutine*.
Nota tecnica: dato un ISA questo può essere servito da ABI diversi, poiché l'ABI influenza solo la gestione dei servizi.