Fondamenti d'Informatica

Introduzione all'architettura dei calcolatori

F. Fabris

Bozza del 11 dicembre 2020

Corso di Laurea Triennale in *Ingegneria Elettronica e Informatica*

Indice

Prefazione

Questa dispensa contiene il materiale didattico necessario per la prima parte del corso di *Fondamenti d'Informatica*, inserito al I anno della laurea triennale in *Ingegneria Elettronica e Informatica*, presso l'Università degli Studi di Trieste. Essa è stata pensata come un'introduzione all'architettura dei calcolatori e alle tecnologie elettroniche che la supportano.
Si parte da un'introduzione storica all'Informatica (capitolo 1) seguita da una riflessione sul significato dell'informazione (capitolo 2), sottolineando la differenza tra *informazione sintattica* e *informazione semantica*. Si tratta successivamente della *codifica* dell'informazione dal mondo esterno - alfabeti finiti, notazioni numeriche, segnali analogici e immagini - al mondo interno del calcolatore, fatto essenzialmente di stringhe binarie.
Nel capitolo 3 si fornisce una brevissima introduzione alle tecnologie elettroniche, mostrando la storia e i principi di funzionamento dei principali dispositivi elettronici elementari (diodi a vuoto, triodi, diodi a semiconduttore, transistor).
Attraverso lo studio formale dell'*Algebra Booleana* (capitolo 4), autentico pilastro di tutta l'elettronica digitale, si perviene poi alla realizzazione delle *porte logiche*, che vengono caratterizzate anche dal punto di vista circuitale mediante i transistor, in modo da creare una connessione tra funzione astratta e sua attuazione circuitale.
I capitoli 5 e 6 sono dedicati rispettivamente ai *circuiti combinatori* e a quelli *sequenziali*, con l'obiettivo di realizzare i moduli circuitali - registri, contatori, *multiplexer*, sommatori, *flip-flop*, ecc.- che rappresentano le cellule costitutive di base per la realizzazione di un qualunque dispositivo digitale, primo fra tutti il *computer*.
Tutti i temi relativi ai dispositivi elettronici, alle porte logiche e ai circuiti combinatori e sequenziali, saranno ripresi e ampliati, anche nell'ambito di un'attività nei laboratori, nei corsi di *Reti Logiche* ed *Elettronica* e nei corsi di ulteriore specializzazione.
Il capitolo 8 illustra il *modello di computazione* di *Shepherdson-Sturgis* (detto anche modello RAM), che costituisce una chiave di lettura del livello *assembler* della macchina; contemporaneamente esso anticipa i temi che saranno trattati nel corso di *Complessità e Crittografia* del IV anno, dove si ragionerà sui limiti intrinseci dell'approccio procedurale-algoritmico, delineati rispettivamente dalla *Teoria dell Computabilità* (problemi indecidibili) e dalla *Teoria della Complessità* computazionale (problemi intrattabili).
Il capitolo 7 illustra invece, in modo sintetico, la struttura architetturale generale di un calcolatore, soffermandosi sull'organizzazione della CPU, sull'esecuzione del ciclo *fetch-execute* e sulla gerarchia di memoria.

Lo spirito col quale è stato organizzato il materiale è soprattutto quello di sviluppare, nell'allievo ingegnere, l'idea che la straordinaria domanda di complessità che caratterizza i progetti odierni dei computer trova risposta nel rigore metodologico dell'astrazione matematica in generale, e dell'Algebra Booleana in particolare; per questo motivo si è dedicato a quest'ultima disciplina uno spazio maggiore del solito, in modo che sia molto solida la base sulla quale erigere le infrastrutture e le gerarchie circuitali di ordine superiore. Si sono viceversa trascurati alcuni particolari legati allo sviluppo tecnologico e/o al dettaglio fine dell'architettura, che andrebbero eventualmente affrontati in un corso avanzato di architetture.

La maggior parte del materiale didattico riguardante la parte centrale e più significativa del corso (Algebra Boolena, porte logiche, circuiti combinatori e sequenziali, moduli circuitali) è stata organizzata sulla base delle dispense realizzate a suo tempo dal prof. Antonio D'Amore [1], che per moltissimi anni è stato docente dei corsi di *Circuiti logici e impulsivi* e di *Reti logiche* presso l'Università degli Studi di Trieste; sono stati molto utili anche

i testi di Bucci [2] e di Clements [3], ai quali rimandiamo i lettori che volessero approfondire gli aspetti tecnici più sofisticati che in questa sede non si sono potuti affrontare.

Il materiale trattato nel corso avrà il suo naturale completamento nei corsi di *Reti Logiche* (II anno) e in quello di *Sistemi Operativi* (III anno per il curriculum di *Applicazioni Informatiche*) della Laurea Triennale in Ingegneria Elettronica e Informatica.

Capitolo 1

Introduzione storica

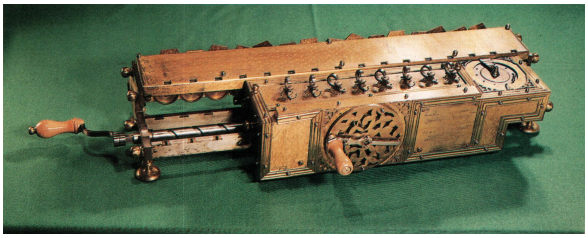
Quando si pensa alla parola *Informatica*, che deriva dalla contrazione francese di *Informa*tion Automa*tique*, l'immagine corre necessariamente al calcolatore e ai suoi accessori periferici (la tastiera, lo schermo, il *mouse* ecc.), al punto che la traduzione inglese viene resa con la locuzione *Computer Science*. Anche se è inevitabile riconoscere l'importanza del calcolatore, inteso come macchina fisica che attua gli schemi concettuali evocati dall'Informatica e che ha decretato il successo e la permeabilità delle tecnologie informatiche, è necessario prendere coscienza del fatto che l'Informatica non è *riducibile* alla macchina. Essa non solo è indipendente dalla tecnologia specifica impiegata per costruire i calcolatori (nella fattispecie la tecnologia elettronica dei semiconduttori), ma è indipendente persino dall'esistenza di una macchina fisica che la renda operativa, tant'è che i fondamenti dell'Informatica, dati dalla *Teoria della Computabilità*, furono sviluppati *prima* della costruzione materiale del primo calcolatore digitale, lo *Z1*, attuata dall'ingegnere tedesco *Konrad Zuse* tra il 1936 e il 1938.

La tecnologia informatica si sviluppa a partire dalla metà degli anni '30, in un momento felice di congiunzione tra due correnti operative e di pensiero ben distinte: da una parte c'era chi inseguiva il sogno millenario di una macchina per fare i calcoli in modo automatico (meccanica nelle prime versioni, elettromeccanica ed elettronica nelle ultime); dall'altra c'era chi si occupava dei *fondamenti logici e assiomatici della Matematica*, sognando una sorta di "meccanizzazione" della stessa, che consentisse di ricavare tutti i *teoremi* di una certa teoria matematica a partire dai suoi *assiomi* e dalle *regole di inferenza*. L'interazione tra queste due correnti di pensiero costituì il contesto fecondo attraverso il quale si passò dai sogni alla realtà.

1.1 Dalle calcolatrici meccaniche al primo computer

La storia della computazione numerica parte dagli abaci cinesi del 1200 D.C., mentre la prima realizzazione di una macchina automatica per il calcolo aritmetico viene fatta risalire a *Blaise Pascal*, filosofo, matematico e fisico francese, che nel 1643 realizzò un dispositivo meccanico per eseguire automaticamente addizioni e sottrazioni, la cosiddetta *Pascalina* (fig.1.1a). È però acclarato che già 150 anni prima *Leonardo da Vinci* aveva progettato una macchina analoga, anche se non arrivò mai a una sua costruzione. Qualche anno dopo, a partire dal 1674, il famoso filosofo e matematico tedesco *Gottfried Wilhelm Leibniz* presentò il progetto di una macchina calcolatrice a ruote e ingranaggi (le *Ruote di Leibniz*) , che era in grado di effettuare moltiplicazioni e divisioni (si veda figura 1.1b). Leibniz è però famoso soprattutto per il suo contributo fondamentale all'individuazione delle basi della Logica Simbolica ("*L'Arte Combinatoria*"), su cui si regge il funzionamento di moderni calcolatori. I successivi sviluppi in tale settore, ad opera di *George Boole*, *Alfred Whitehead*, *Bertrand Russell* e *Giuseppe Peano*, diedero consistenza al sogno di Leibniz di un ragionamento simbolico universale, con la nascita di una nuova disciplina matematica, la *Logica Simbolica*. L'idea di fondo dell'*Arte Combinatoria* è quella di trovare una logica capace non

(a) La macchina *Pascalina* per fare somme e sottrazioni

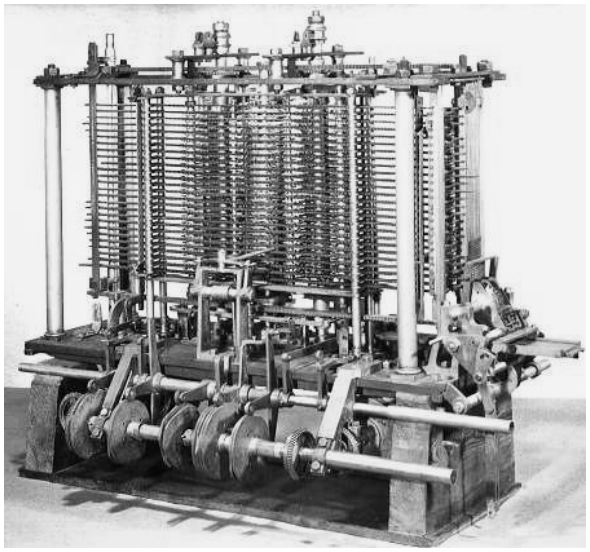


(b) Le *Ruote di Leibniz* per effettuare moltiplicazioni e divisioni

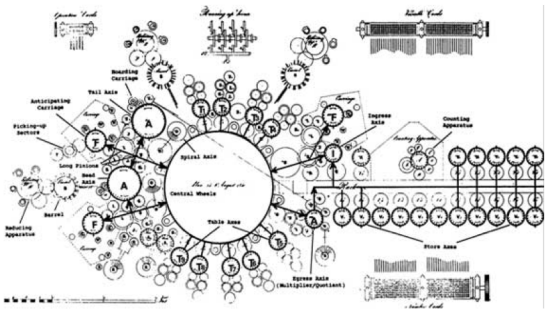
Figura 1.1: Le prime macchine calcolatrici di tipo meccanico

soltanto di dimostrare la verità di ogni proposizione vera, ma anche di costruire nuove proposizioni con la certezza dei procedimenti matematici.

Il primo modello di calcolatore così, come noi lo intendiamo oggi, che fosse cioè in grado di manipolare non solo numeri, ma anche simboli, lo si deve a *Charles Babbage*, matematico, filosofo e ingegnere britannico, il quale descrisse nel 1834 il progetto della *Macchina Analitica*, modello per tutti i successivi calcolatori digitali universali (fig.1.2). La macchina non fu mai realizzata per le difficoltà legate alla complessità meccanica delle



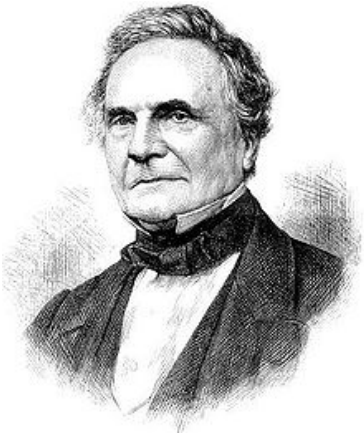
(a) Ricostruzione della *Macchina Analitica*



(b) Schema del 1840 della *Macchina Analitica*

Figura 1.2: La *Macchina Analitica* dell'ingegnere inglese *Charles Babbage*

sue 25 mila parti, anche perché i concetti sui quali avrebbe basato il suo funzionamento anticipavano di almeno cent'anni il livello tecnologico necessario alla loro attuazione pratica. Per questa macchina egli aveva infatti immaginato la possibilità di introdurre da un lato le "regole" della computazione (che noi oggi chiameremmo *algoritmi*) e dall'altro i valori da associare alle variabili e alle costanti, e tutto ciò impiegando schede o nastri perforati del tutto simili a quelli usati nei telai tessili di *Jacquard* fin dai primissimi anni dell'ottocento. I concetti che stanno alla base della *Macchina Analitica* sono gli stessi usati oggi per i moderni calcolatori elettronici. La macchina era costituita da due parti: la memoria (*Store*) che immagazzinava variabili e costanti e nella quale erano conservati anche tutti i risultati intermedi dei calcoli; l'unità di calcolo (*Mill*) che conteneva il programma vero e proprio. Lo schema generale del suo calcolatore è talmente simile a quello dei computer moderni che la tardiva riscoperta dei



(a) Charles Babbage

(b) Ada Byron contessa Lovelace

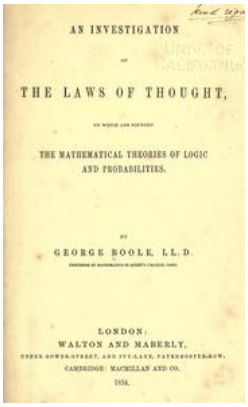
(c) Medaglia commemorativa della *Association for Computing Machinery* (ACM)

Figura 1.3: Charles Babbage e Ada Byron

suoi scritti invalidò alcuni brevetti della IBM. L'opera di Babbage venne poi esaltata da una singolare nobildonna inglese, *Ada Byron*, contessa di *Lovelace* (figlia del poeta Lord Byron), che per prima intuì l'universalità delle idee espresse da Babbage. Tra i due iniziò un fitto scambio di lettere, piene di numeri, idee, fatti e fantasie e nel 1843, in uno scritto ormai famoso, Ada Byron descrisse le possibili applicazioni della macchina nel calcolo matematico, ipotizzando persino il concetto di *Intelligenza Artificiale* e affermando che la macchina, quando realizzata, sarebbe stata cruciale per il futuro della scienza. A titolo di esempio spiegò il modo in cui la macchina avrebbe potuto effettuare il calcolo dei *numeri di Bernoulli*, e così facendo scrisse quello che viene unanimemente riconosciuto come il primo *programma per computer* della storia. In onore di Ada Byron, nel 1979 il Dipartimento della Difesa degli Stati Uniti battezzò *Ada* un linguaggio di programmazione che era stato appena realizzato.

Nel frattempo *George Boole* (fig.1.4a), logico e matematico britannico, cominciò a lavorare sullo strumento concettuale che sta alla base del funzionamento dei moderni calcolatori, cioè la logica binaria, o *Logica Booleana*, scrivendo l'opera "*An investigation of the Laws of Thought*" (fig (1.4b)). Si tratta di un calcolo logico a due valori

(a) George Boole


AN INVESTIGATION
OF
THE LAWS OF THOUGHT,
ON WHICH ARE FOUNDED
THE MATHEMATICAL THEORIES OF LOGIC
AND PROBABILITIES.
BY
GEORGE BOOLE, LL.D.
LONDON:
WALTON AND MABERLY,
CAMBRIDGE: MACMILLAN AND CO.

(b) Il frontespizio dell'opera *An Investigation on the Law of Thought*, di *George Boole*

AND			OR			NOT	
X	Y	Z = X·Y	X	Y	Z = X+Y	X	Z = $\overline{X}$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

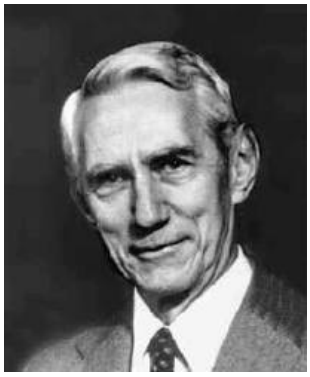
(c) Le tabelle di verità per le funzioni Booleane AND, OR e NOT

Figura 1.4: George Boole, il padre della Logica Booleana

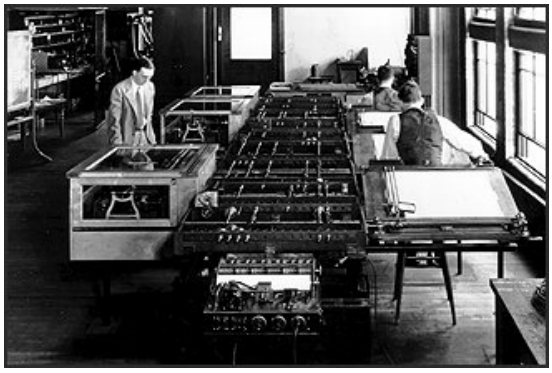
di verità, *Vero* e *Falso*, che consente di operare su proposizioni allo stesso modo in cui si opera su entità matemati-

che. Nel suo lavoro Boole mostrò che la logica Aristotelica può essere rappresentata tramite equazioni algebriche. Boole sviluppò i concetti precedentemente espressi da Leibniz sul sistema binario e descrisse gli operatori logici che da lui presero poi il nome di *Operatori Booleani* (AND, OR, NOT), oggi attuati circuitalmente mediante le cosiddette *porte logiche*.

Il lavoro di Boole fu considerato però d'interesse solo matematico-speculativo, almeno fino al 1937, anno in cui *Claude Elwood Shannon* (fig.1.5a), matematico e ingegnere americano, pubblicò la sua tesi di master intitolata *A Symbolic Analysis of Relay and Switching Circuits*. Shannon stava lavorando al MIT sotto la direzione di *Vannevar Bush*, inventore dell'*Analizzatore Differenziale* (fig.1.5b), il primo calcolatore *analogico* per risolvere equazioni differenziali (1930); in particolare egli era interessato alla teoria e alla progettazione dei complessi circuiti di relay che controllavano le operazioni della macchina di Bush.



(a) Claude Elwood Shannon



(b) L'*Analizzatore Differenziale* di *Vannaver Bush* del MIT

Figura 1.5: Claude Elwood Shannon nel laboratorio del MIT

Fu in questo contesto che si rese conto che la Logica Booleana, così come si applicava alla rappresentazione di *Vero* e *Falso*, poteva essere usata per rappresentare le funzioni degli interruttori nei circuiti elettrici, il cui funzionamento è caratterizzato da due stati, acceso e spento. Ciò divenne la base della progettazione dell'elettronica digitale, con applicazioni pratiche nella commutazione telefonica e nell'ingegneria dei computer. I meriti di Shannon vanno però ben oltre, poiché il suo nome è indissolubilmente legato ai due celeberrimi articoli "*A Mathematical Theory of Communications*" del 1948, e "*Communication Theory of Secrecy Sysytems*" del 1949, che gettarono le fondamenta della *Teoria dell'Informazione* e della *Crittografia* moderna.

(a) Archimedes

(b) Brunsviga

(c) Curta

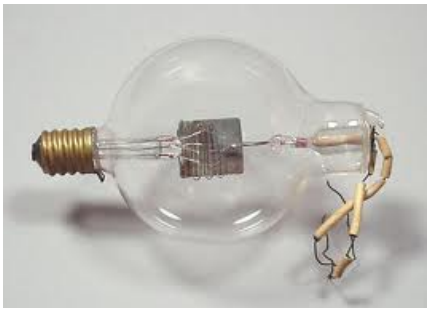
Figura 1.6: Alcuni modelli di macchina calcolatrice meccanica di fine 800, inizi 900

Alla fine dell'ottocento le calcolatrici erano ancora meccaniche, e non essendo stata ancora realizzata la Macchina Analitica prefigurata da Babbage, non esisteva alcuna macchina "programmabile". Nella figura 1.6 ve-

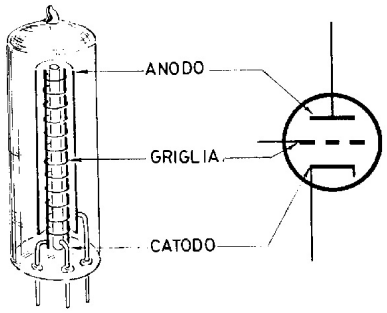
diamo alcuni esempi di calcolatrici in uso all'epoca; tra queste la *Brunsviga*, che ebbe una diffusione notevole e la *Curta*, vero e proprio gioiello nell'arte della meccanica, prodotta fino al 1943.

I primi anni del '900 furono determinanti per il trapasso tra la tecnologia elettromeccanica e quella elettronica dei *tubi termoionici*, che nasce con l'invenzione nel 1904 del *diodo a vuoto* (fig.1.7a), ad opera dell'ingegnere inglese *Sir John A. Fleming*; due anni più tardi l'americano *Lee de Forest*, aggiungendo un terzo elettrodo al diodo di Fleming, la *griglia*, crea il primo *triodo* a vuoto (fig.1.7b), che consente di amplificare un segnale analogico, ma anche di fungere da interruttore comandato in tensione (senza dispendio di potenza), sostituendo così i lenti e pesanti *relay* elettromeccanici, che necessitano per altro di una rilevante potenza per il controllo. La strada è segnata, anche se l'impatto della nuova tecnologia nell'ambito delle macchine da calcolo non sarà immediato, a causa dei problemi di affidabilità ancora presenti.

(a) Diodo a vuoto di *Fleming* (1904)



(b) Triodo a vuoto di *De Forest* (1906)

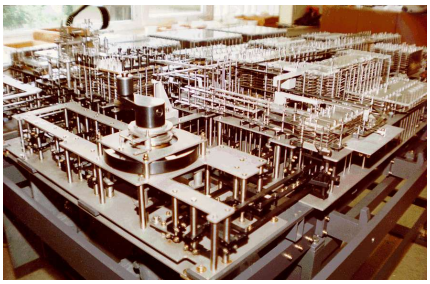


(c) Struttura schematica del triodo

Figura 1.7: I tubi termoionici inventati nei primi anni del 900

Il 1936 è l'anno in cui l'ingegnere tedesco *Konrad Zuse* (fig1.8a) inizia la costruzione del primo calcolatore moderno, la macchina logica "*V1*", successivamente ribattezzata "*Z1*" per evitare qualsiasi riferimento ai tristemente noti razzi V1 tedeschi (fig1.8b). Si tratta di un calcolatore meccanico realizzato artigianalmente e con mezzi rudimentali dallo stesso Zuse, nella propria abitazione (fig1.8c). Il prototipo rappresenta la prima macchina al mondo, basata su codice binario, completamente programmabile. Zuse, convinto che i programmi composti da combinazioni di bit potessero essere memorizzati, chiese anche un brevetto in Germania per l'esecuzione automatica di calcoli.

(a) L'ingegnere tedesco *Konrad Zuse*, che costruì il primo calcolatore nel 1936



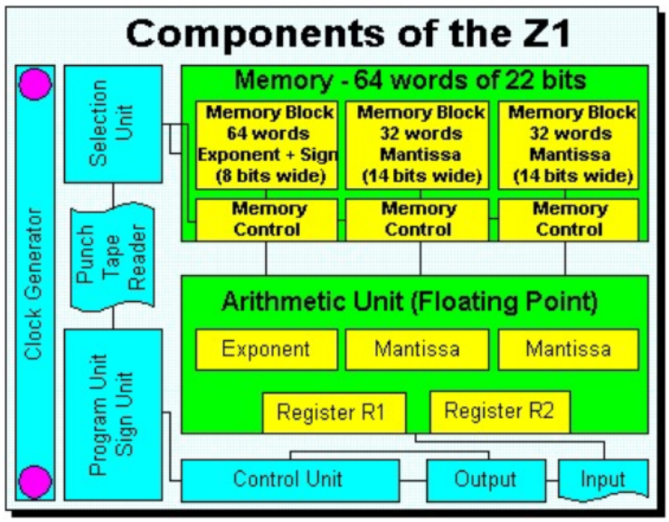
(b) Il primo calcolatore del mondo, lo Z1, del 1937

(c) Soggiorno della casa di Zuse dove venne costruito lo Z1

Figura 1.8: Lo Z1 venne distrutto subito dopo la costruzione, a seguito di un bombardamento di Berlino

Lo *Z1* era un apparecchio programmabile, in grado di processare numeri in formato binario e le cui caratteristiche più apprezzabili, viste con il senno di poi, furono la netta distinzione fra memoria e processore. Questa architettura (fig1.9a), che non venne adottata dall'*ENIAC* o dal *Mark I*, (i primi computer realizzati negli Stati Uniti

quasi dieci anni più tardi), rispecchia l'architettura del calcolatore ipotizzata solo nel 1945 da *John von Neumann*. Lo *Z1* conteneva tutti i componenti di un moderno computer, anche se era completamente meccanico, come ad esempio le unità di controllo, la memoria, la rappresentazione a virgola mobile, ecc. Aveva una frequenza di lavoro di 1 *Hertz*, era in grado di effettuare una moltiplicazione in 5 secondi, disponeva di 64 celle di memoria a 22 bit e usava al posto dei *relay* circa 20.000 piastre in metallo (fig.1.9b). Il calcolatore venne poi distrutto assieme ai progetti dai bombardamenti di Berlino, durante la seconda guerra mondiale, ma nel 1941 venne costruita la sua terza versione, denominata *Z3* (figura 1.10a), che diventerà operativo per qualche tempo, prima di essere a suo volta distrutto da un bombardamento.



(a) Architettura dello Z1

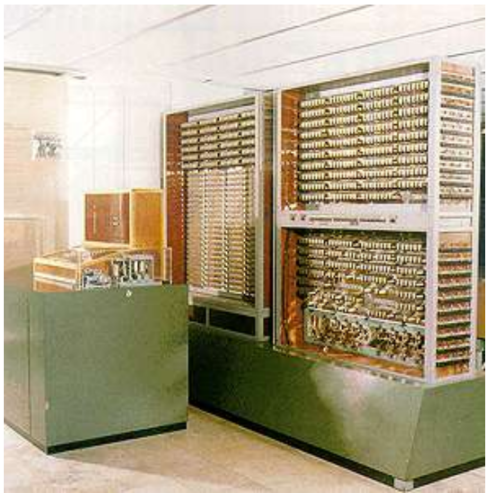
Data sheet

Name of the machine	Z1
Implementation	Thin metal plates, worked with fret saw
Frequency	1 Hertz
Numeric unit	Floating point unit, 22 Bit word length
Average calculation speed	Multiplication approx. 5 seconds
Input	Decimal keyboard, automatic binary coding
Output	Decimal digits
Word length	24 Bit mantissa, 8 bit exponent, 1 sign
Number of relays	No relays, thousands of metal plates, approx. 20,000 parts
Memory	64 cells à 22 Bit
Power consumption	Approx. 1000 watts for the electric cycle motor
Weight	Approx. 500 kg
Area of application	Experimental model, no application, developed for scientific calculations
Number of machines sold	0
Cost in DEM	No price
Comments	The Z1 was not reliable. A

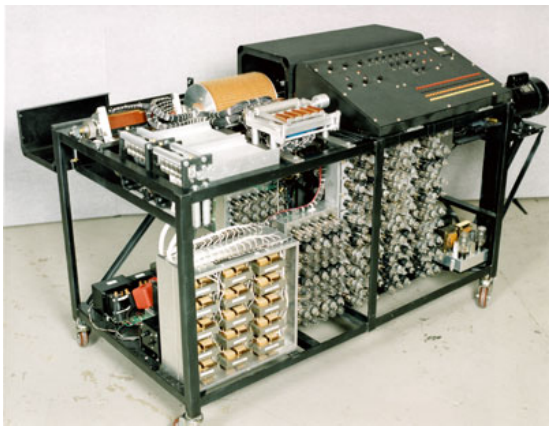
(b) Le principali caratteristiche tecniche dello Z1

Figura 1.9: Architettura e principali caratteristiche dello Z1, basato su una tecnologia puramente meccanica

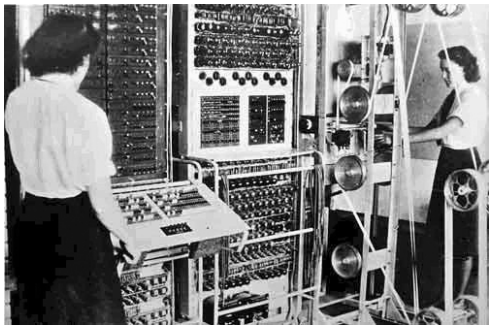
Negli Stati Uniti inizia nel 1939 il progetto dell'*Automatic Sequence Controlled Calculator* (*ASCC*) della IBM, che in seguito verrà ceduto all'università di Harvard e prenderà il nome di *Mark I* (fig.1.10d). Quasi contemporaneamente parte anche il progetto del calcolatore "*ABC*" di *J.V. Atanasov* e *C. Berry* (fig.1.10b), sul quale si sarebbe basato successivamente *J.W. Mauchly* per la costruzione dell'*ENIAC* (fig.1.10e). L'ABC è il primo computer che utilizza la nuova tecnologia dei *tubi termoionici* (o a vuoto). Il prototipo, che realizza somme a 16-bit, non arriverà mai in produzione, ma i concetti contenuti nell'*ABC*, come l'*Unità Aritmetico Logica* (ALU) e la memoria riscrivibile, compariranno nei moderni computer. Negli ultimi anni ci sono state molte controversie su chi avesse veramente inventato il primo computer elettronico digitale, e una corte di giustizia americana decise in favore di Atanasov. Nel Regno Unito si costruisce invece, nel 1943, il *Colossus* (fig.1.10c), progettato per poter forzare i cifrari tedeschi basati sull'impiego della macchina cifrante di Lorenz SZ40/42, in uso al comando generale del III Reich. L'ultimo calcolatore d'interesse storico che menzioniamo è l'EDVAC (fig.1.10f), che fu progettato all'inizio da John von Neumann, il matematico e ingegnere ungherese cui si deve l'odierna architettura dei computer, basata su un *Unità Logico-Aritmetica* (ALU), su dei *registri* di memoria e su una *memoria RAM* per memorizzare i dati e il programma. Diversamente dal suo predecessore decimale ENIAC, l'EDVAC era binario.



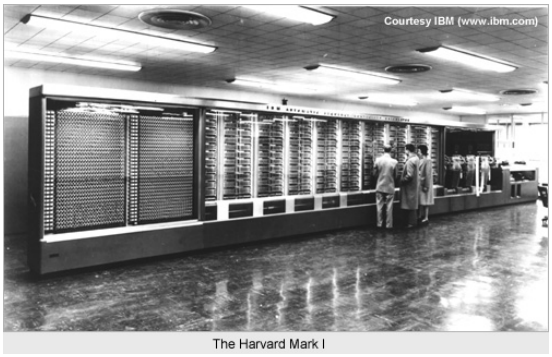
(a) Il calcolatore Z3 di Zuse (Maggio 1941), con un'architettura simile a quella di von Neumann, era basato sulla tecnologia elettromeccanica dei relè



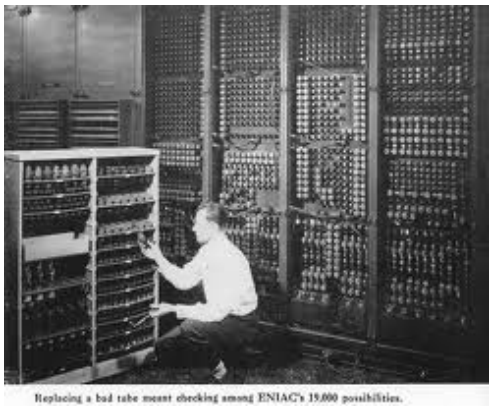
(b) Il primo calcolatore a tubi termoionici, l'ABC di *Atanasov* e *Berry* (Estate 1941)



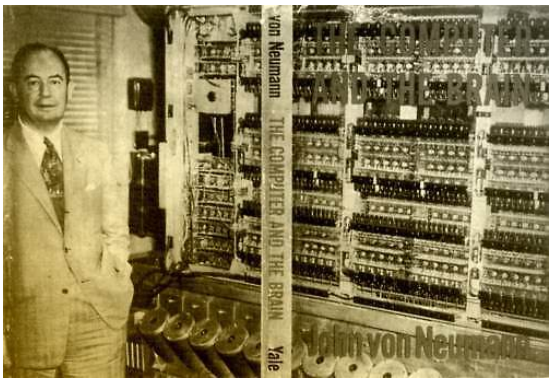
(c) Il calcolatore *Colossus*, prodotto in Inghilterra nel 1943 e usato per la decifrazione della macchina cifrante it Lorenz, in uso al quartier generale di Hitler



(d) Il *Mark I* fu il completamento del progetto *ASCC* realizzato dall'università di Harvard (1944), ma era ancora basato sulla tecnologia elettromeccanica dei relè



(e) Il calcolatore *ENIAC*, basato sulla tecnologia a tubi termoionici dell'ABC (1946-48)



(f) L'EDVAC fu uno dei primi calcolatori con l'architettura ideata da *von Neumann* (1951)

Figura 1.10: I calcolatori del periodo 1941-1951, realizzati da Germania, Regno Unito e USA

1.2 Dal programma di Hilbert ai teoremi di incompletezza di Gödel

L'ideazione e la realizzazione delle prime macchine calcolatrici, secondo il processo storico delineato brevemente nel paragrafo precedente, ebbe come spinta propulsiva la necessità di effettuare in modo automatico le quattro operazioni elementari con i numeri (somme, moltiplicazioni, differenze e divisioni). Tuttavia la complessità strutturale via via crescente di tali macchine, che ebbero come capostipite la *Macchina Analitica* di Babbage, trasformò completamente la loro natura: infatti il "programma" di calcolo, inizialmente incarnato negli ingranaggi meccanici o nei circuiti elettromeccanici delle macchine più avanzate, e deputato alla soluzione di un problema specifico, lascia a un certo punto il posto a un "programma" non cablato, che può essere modificato dall'esterno con lo scopo di poter risolvere un problema nuovo, e ciò senza dover riassemblare la macchina. La macchina acquista dunque una flessibilità che le consente di essere usata più volte per risolvere problemi di natura diversa, e ciò senza dover cambiare la sua topologia. Diventa cioè una *macchina programmabile*. All'inizio questi problemi erano legati soprattutto al calcolo di fattori numerici, ma il potere della *codifica simbolica*, ossia la libertà di attribuire un qualunque significato a un simbolo, coniugato con la possibilità di manipolare i simboli in modo logicamente strutturato, portò a disvelare potenzialità inizialmente insospettabili per le macchine sia pur rudimentali dell'epoca.

È a questo punto che il destino di coloro che inseguivano il sogno di una macchina automatica per fare i calcoli s'intreccia con quello di coloro che miravano a una ricostruzione logica e unitaria di tutta quanta la Matematica, che avrebbe dovuto consentire di ricavarne i teoremi in qualsiasi ambito (analisi, geometria, algebra, ecc.) a partire dagli *assiomi* e dalle *regole di inferenza*, secondo un approccio che si inquadra perfettamente col pensiero razionalista e riduzionista di inizio secolo.

Ricordiamo a tal proposito che ogni *Teoria Matematica*, quale ad esempio l'*Aritmetica*, la *Teoria degli Insiemi* ecc., scaturisce da alcune affermazioni iniziali considerate vere e denominate *assiomi* (o *postulati*), e dall'applicazione ad essi di specifiche *regole di inferenza*, che esprimono le modalità lecite per costruire altre affermazioni vere, denominate *teoremi*. In quest'ottica la *Teoria* è l'insieme di tutti gli assiomi, che giocano il ruolo di verità primitive, e di tutti i teoremi che si possono provare usando le regole di inferenza. Ecco ad esempio i celebri *postulati di Peano*, sui quali si fonda l'Aritmetica dei numeri naturali:

Postulati di Peano

1. "0" è un numero;
2. se n è un numero, il suo successore è un numero;
3. "0" non è successore di alcun numero;
4. numeri diversi non possono avere lo stesso successore;
5. se un insieme S di numeri comprende lo 0, come anche il successore di qualunque numero in S, allora S comprende tutti i numeri.

Per quanto riguarda le regole di inferenza, possiamo dire che esse costituiscono i cardini logici del ragionamento matematico; alcuni esempi sono la *Modus Ponens*, la *Modus Tollens* e la *Reductio ad Absurdum*, illustrate sinteticamente dalla tabella sotto riportata. La barra indica che a partire dalle premesse che stanno sopra, si trae la conseguenza che sta sotto:

Modus Ponens	$\frac{A \to B,\ A}{B}$	
Modus Tollens	$\frac{A \to B,\ nonB}{nonA}$	Il *Modus Ponens* si legge così: se da A segue B, e A
Reductio ad Absurdum	$\frac{A \to B,\ A \to nonB}{nonA}$	

è vero, allora è vero anche B; analogamente le altre.

Il rappresentante sommo dell'impostazione riduzionista prima citata fu il grande matematico tedesco *David Hilbert* (1862–1943, fig1.11a). Al *Secondo Congresso Internazionale di Matematica* di Parigi del 1900, Hilbert tenne un intervento di portata storica - probabilmente la più influente conferenza matematica di ogni tempo - proponendo un elenco di 23 problemi aperti che, a suo giudizio, costituivano una sfida per i matematici del secolo a venire. La tabella di figura 1.12 riporta l'elenco completo. La natura di questi problemi era varia e disomogenea: se

(a) David Hilbert

(b) Kurt Gödel

(c) Alan Turing

Figura 1.11: La formulazione dei 23 problemi di Hilbert consentì a Gödel e a Turing di sviluppare le riflessioni che portarono alla fine alla formulazione del primo modello di computazione nel 1936

alcuni erano molto specifici e tecnicamente ben delineati (p.es. il Problema 3, che venne risolto immediatamente), altri (p.es. il Problema 6, sull'assiomatizzazione della Fisica, o il Problema 4) erano troppo generali o troppo vaghi per ammettere una risposta incontrovertibile. Altri ancora, i Problemi 1, 2 e 10, portarono a una soluzione inaspettata per Hilbert: essi ci riguardano da vicino, per la loro stretta connessione con i fondamenti della *Teoria della Computabilità*, e quindi con un inquadramento formale dei fondamenti dell'Informatica. Data la loro importanza li esaminiamo a parte.

1^o Problema - Ipotesi del continuo (IC)

Non esiste una cardinalità intermedia tra quella dei naturali e quella dei reali.

Si tratta di accertare se esista un insieme S (infinito) dotato di cardinalità inferiore a quella dei reali e superiore a quella dei naturali. Nel 1938 *Kurt Gödel* stabilì che IC non è refutabile nell'ambito assiomatico della teoria (di Zermelo-Fraenkel) degli insiemi; nel 1963, *Paul Cohen* stabilì che in tale ambito non è neppure dimostrabile.

2^o Problema - Assiomi dell'aritmetica

Accertare che gli assiomi dell'aritmetica sono consistenti.

Gödel dimostrò (1931, *1^o Teorema di Incompletezza*) che in qualunque sistema assiomatico sufficientemente espressivo da contenere almeno l'aritmetica, si può individuare un enunciato circa i numeri naturali che:

non può essere nè dimostrato nè refutato all'interno del sistema (sistema *incompleto*);
oppure
può venire sia provato che refutato all'interno del sistema (sistema *inconsistente*).

Problema	1	risolto (1963)	*L'Ipotesi del Continuo*
Problema	2	risolto (1930)	*Consistenza degli assiomi dell'aritmetica*
Problema	3	risolto	Uguaglianza di volumi tra tetraedri
Problema	4	troppo vago	Costruzione di tutte le metriche con linee geodetiche
Problema	5	risolto	Differenziabilità dei gruppi continui di trasformazioni
Problema	6	aperto	Assiomatizzazione della Fisica
Problema	7	risolto	Trascendenza di a^b, con $a \neq 0, 1$ e b irrazionale
Problema	8	aperto	Ipotesi di Riemann e congettura di Goldbach
Problema	9	parzialm. risolto	Trovare la legge più generale di reciprocità in un qualunque campo algebrico numerico
Problema	10	risolto (1970)	*Risolubilità delle equazioni Diofantee*
Problema	11	parzialm. risolto	Forme quadratiche con coefficienti numerici algebrici
Problema	12	aperto	Estensioni di campi numerici algebrici
Problema	13	risolto	Risoluzione di equazioni di 7-imo grado usando funzioni di due argomenti
Problema	14	risolto	Dimostrazione di finitezza di certi sistemi completi di funzioni
Problema	15	parzialm. risolto	Fondamenti del calcolo enumerativo di Schubert
Problema	16	aperto	Topologia di curve e superfici algebriche
Problema	17	risolto	Espressione di funzioni razionali definite come quozienti di somme di quadrati
Problema	18	risolto	Riempimento spaziale tramite poliedri non regolari
Problema	19	risolto	Analiticità delle soluzioni di Lagrangiani
Problema	20	risolto	Risolvibilità di ogni problema variazionale fissate certe condizioni al contorno
Problema	21	risolto	Esistenza di equazioni differenziali lineari aventi un gruppo monodromico assegnato
Problema	22	risolto	Uniformizzazione di relazioni analitiche per mezzo di funzioni automorfiche
Problema	23	risolto	Sviluppi ulteriori del calcolo variazionale

Figura 1.12: I 23 Problemi di Hilbert

In altre parole *ogni sistema assiomatico sufficientemente espressivo è o inconsistente o incompleto*. Se escludiamo la prima eventualità, possiamo esprimere il teorema più semplicemente, dicendo che non è detto che un enunciato vero sia un teorema, e cioè che discenda dagli assiomi tramite le regole di inferenza del sistema.
Da qui Gödel dedusse (*2^o Teorema di Incompletezza*) che quando un sistema assiomatico è consistente e sufficientemente espressivo da contenere almeno l'aritmetica, esso *non può* provare la propria consistenza. Ciò fornisce una risposta, per quanto negativa e imprevista, al 2^o problema di Hilbert.

10^o Problema - Risolubilità delle equazioni Diofantee.

Trovare una procedura in grado di stabilire se una qualunque equazione Diofantea assegnata ammette soluzioni intere.

Un'equazione Diofantea è un'equazione polinomiale $p(x_1, x_2, ..., x_n) = 0$ a coefficienti interi, che s'intende risolvere assegnando valori interi alle incognite x_i. *Yuri Matiyasevich* dimostrò, nel 1970, che una procedura risolutiva generale non può esistere: a meno che non si pongano fortissime limitazioni al numero di incognite o al grado del polinomio; siamo dunque di fronte a un ulteriore importante problema della matematica che viene risolto in senso tanto negativo quanto inaspettato.

È evidente che l'impostazione riduzionista di Hilbert, implicita per altro già negli enunciati dei problemi (nei quali si chiede di trovare *la* soluzione, e non *se* una certa soluzione esista o meno), subì un duro e inaspettato contraccolpo dall'enunciazione dei *Teoremi di Incompletezza* di Gödel, che rimangono sicuramente una delle più importanti scoperte matematiche del '900. Essi gettarono lo scompiglio tra le fila dei matematici dell'epoca, poiché l'idea che qualcosa di matematicamente vero potesse non esser dimostrabile implicava un ridimensionamento essenziale, anche se circoscritto a singoli problemi, nella capacità argomentativa del metodo matematico basato sull'approccio *ipotetico-deduttivo*. D'altra parte il programma riduzionista, chiaramente perseguito anche dai matematici *Gottlob Frege* e *Giuseppe Peano*, aveva già subito qualche incrinatura, soprattutto ad opera di *Bertrand Russell* e in singolare contemporaneità con l'elencazione dei 23 problemi di Hilbert. Il suo famoso *paradosso* aveva destabilizzato l'opera di *Frege*, aprendo un periodo di crisi dei fondamenti logici della matematica. Tale paradosso riguarda *gli insiemi che non sono membri di sé stessi*. A prima vista la loro stessa definizione potrebbe sembrare mal posta; e in effetti, se prendiamo come riferimento un insieme di numeri, esso non è un numero, per cui sembra privo di senso chiedersi se appartenga o meno a sé stesso. Tuttavia,, tanto per fare un esempio, l'insieme degli argomenti trattati in questo paragrafo è esso stesso un argomento (ne stiamo parlando ora!), e dunque è un insieme che appartiene a sé stesso.

L'antinomia di Russell

Se chiamiamo T l'insieme di tutti gli insiemi che non appartengono a sé stessi si ha:

se $T \in T$ allora $T \notin T$, poiché T contiene per definizione solo insiemi che *non appartengono a sé stessi*

se $T \notin T$ allora $T \in T$, poiché T contiene per definizione tutti gli insiemi che *non appartengono a sé stessi*

Il paradosso aveva famosi precedenti storici, quali il *Paradosso del mentitore*, attribuito ad *Eubulide di Mileto* (filosofo greco del IV secolo A.C.): *Un uomo dice che sta mentendo. Sta dicendo il vero o il falso?* Di questo paradosso è nota anche la variante *Questa frase è falsa*, e una sua versione precedente, attribuita ad *Epimenide*, cretese: *I cretesi son tutti bugiardi*, che non sembra però essere stata scritta con l'intento di illustrare un paradosso. Tuttavia il *Paradosso del mentitore* è una contraddizione logica che gioca sull'autoreferenzialità in un contesto, come quello linguistico, che non è formalizzato matematicamente; infatti la spiegazione più semplice consiste nell'assumere che ogni frase pronunciata (o scritta) esprima implicitamente un'affermazione di verità sull'oggetto della frase stessa, per cui la frase *Questa frase è falsa* andrebbe letta in realtà come *Questa frase è vera e questa*

frase è falsa, il che corrisponde all'enunciazione di una semplice contraddizione del tipo A *e non* A, che è falsa. Nel caso del paradosso di Russell le cose erano invece molto più compromesse: il suo argomento evidenziava che una teoria matematica proposta come fondamentale, la *Teoria Elementare degli Insiemi* di *Cantor* nell'assetto formale raggiunto a fine '800, era minata da irriducibili contraddizioni interne.

Nel 1908 *Ernst Zermelo* riuscirà a sanare l'antinomia di Russell, impostando un nuovo sistema noto oggi come *Teoria assiomatica degli Insiemi di Zermelo-Fraenkel*; ma con l'effetto destabilizzante causato dai teoremi di Gödel, il programma Hilbertiano, teso alla riorganizzazione di tutta la matematica in un edificio formale che avrebbe dovuto autocertificare la propria consistenza, dovrà venire definitivamente archiviato.

1.3 La nascita dell'Informatica

Sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica sopra evocati, si sviluppò quella corrente di pensiero che riuscì in seguito a delineare il nucleo fondante dell'Informatica, cioè la *Teoria della Computabilità*, intesa come studio, modellizzazione e individuazione dei limiti relativi all'approccio computazionale basato sulle *procedure effettive* o *algoritmi*. Di nuovo lo spunto iniziale partì da Hilbert, che nel 1928 scrisse, con W. Ackermann, il libro "*Grundzüge der theoretischen Logik*"; in quest'opera compare per la prima volta l'enunciazione del famoso *Entscheidungsproblem* (Problema della decisione) per la *Logica dei predicati (del Primo Ordine)*, cioè per il sistema formale che incorpora la logica classica basata sugli operatori *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implica* ($\Longrightarrow$), *per ogni* ($\forall$), *esiste* ($\exists$). Per capire il significato del *Entscheidungsproblem*, ricordiamo che in tale sistema formale si possono formare delle formule, le cosiddette *formule ben formate*, come per esempio

$$(\exists F)\{(F(a) = b) \wedge (\forall x)[p(x) \implies (F(x) = g(x, F(f(x))))]\}$$

che si legge come "esiste una funzione F tale che $F(a) = b$ e tale che $\forall x$, se è vero il predicato $p(x)$, allora $F(x) = g(x, F(f(x)))$". Ciascuna formula è suscettibile di una *interpretazione*, che consiste nell'assegnazione delle funzioni, delle variabili e delle costanti. Per esempio, assegnando $f(x) = x - 1$ e $g(x) = xy$ sui numeri naturali, l'interpretazione diventa:

Interpretazione 1: $f(x) = x - 1$ e $g(x, y) = xy$

$$(\exists F)\{(F(0) = 1) \wedge (\forall x)[x > 0 \implies (F(x) = xF(x - 1)]\}$$

che si legge come "esiste una funzione F tale che $F(0) = 1$ e tale che $\forall x$, se $x > 0$ allora $F(x) = xF(x - 1)$; tale interpretazione è vera, poiché corrisponde alla funzione fattoriale. Viceversa, l'interpretazione seguente

Interpretazione 2: $f(x) = x$ e $g(x, y) = y + 1$

$$(\exists F)\{(F(0) = 1) \wedge (\forall x)[x > 0 \implies (F(x) = F(x) + 1]\}$$

risulta evidentemente falsa. Una formula si dice allora *valida* se è vera in tutte le interpretazioni. L'oggetto del *Entscheidungsproblem* riguarda proprio la validità delle formule nella logica dei predicati.

Entscheidungsproblem

Trovare una procedura effettiva (algoritmica) per decidere se una qualunque formula nella logica dei predicati è valida (p.es. se una qualunque formula dell'aritmetica è un teorema, cioè derivabile dagli assiomi mediante le regole di inferenza).

Il problema fu risolto indipendentemente da *Alonzo Church*, che pubblicò nel 1936 un articolo intitolato "*An Unsolvable Problem in Elementary Number Theory*", e da *Alan Turing* (fig 1.11c), che nello stesso anno pubblicò

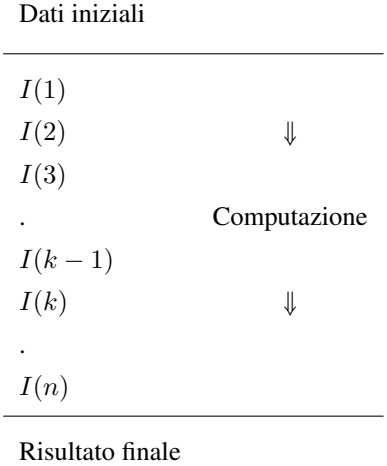
l'articolo "*On Computable Numbers, with an Application to the Entscheidungsproblem*". Essi dimostrarono, con argomentazioni molto diverse, *la non esistenza di un siffatto algoritmo*. Pertanto, in particolare, è impossibile decidere algoritmicamente se un qualunque enunciato sui numeri naturali è vero o meno. Il lavoro di Church fu l'atto di nascita di un formalismo matematico, denominato λ-*calcolo*, che costituisce un vero e proprio *modello di computazione*. Tuttavia l'approccio di Turing, basato su un semplice dispositivo chiamato *macchina di Turing* (MdT), e che oggi riconosciamo come la descrizione del primo modello formale di calcolatore, risultò subito molto più convincente e credibile, al punto che Gödel stesso rimase inizialmente dubbioso sulla correttezza del λ-*calcolo*, ma immediatamente convinto dal modello di Turing. La *Macchina di Turing* incarna implicitamente la prima definizione del concetto di *algoritmo*, al punto che oggi la stretta corrispondenza tra ciò che si considera intuitivamente calcolabile mediante una procedura effettiva di tipo algoritmico e la *Macchina di Turing* costituisce il nucleo forte della cosidetta *Tesi di Church-Turing*. Turing risolse (negativamente, come accennato sopra) l'*Entscheidungsproblem* facendo riferimento al problema della *fermata della macchina di Turing*, e dimostrando che, assegnata una qualunque MdT, non è possibile decidere algoritmicamente se essa si fermerà o meno a partire da certe condizioni iniziali. Il successivo concetto di *macchina di Turing Universale*, cioè di una macchina che sia in grado di simulare la computazione di qualunque altra macchina, getta poi le basi teoriche del calcolatore programmabile.

L'impetuoso sviluppo dell'informatica e il suo rapido affermarsi come disciplina a sé non si può dunque ricondurre solamente alla riflessione millenaria sui limiti del procedimento ipotetico-deduttivo o all'incontro fra questa componente del pensiero matematico e la tecnologia elettronica: la nuova disciplina si sviluppò anche a partire da nuove idee fondanti, la più importante delle quali è proprio quella di *macchina programmabile*, per l'espletamento dei piú diversi compiti senza modifiche della sua architettura fisica. Il sogno di Leibniz di una logica universale, il fervore progettuale di Babbage e i lavori fondamentali di Turing e Church, hanno instradato il pensiero scientifico verso la conquista di un concetto esplicito di *computabilità*, che ha affiancato la realizzazione dei primi calcolatori. Altra idea-chiave, ben manifesta nei progetti di Turing e Zuse, è che l'autoreferenzialità - tradizionale fonte di intriganti paradossi - può essere sfruttata anche in senso positivo. Non c'è una ragione per cui i programmi debbano risiedere all'esterno del calcolatore (come avveniva per i nastri perforati rispetto ai telai Jacquard); un programma caricato in memoria, viceversa, potendo non solo indirizzare le azioni del computer, ma anche subire modifiche per effetto delle sue elaborazioni, avrebbe la duplicità di ruolo necessaria all'apprendimento automatico e alla gestione delle altre tematiche proprie dell'*Intelligenza Artificiale*. Mentre questa seconda idea tarda a dispiegare tutte le potenzialità presenti nella visione di Turing, l'obiettivo di *universalità* del calcolatore può dirsi largamente raggiunto: in effetti, un modesto *laptop* del giorno d'oggi surclassa di molto i colossali calcolatori realizzati da pionieri dell'informatica quali Zuse e von Neumann, che peraltro suscitavano grandi entusiasmi in chi aveva la consapevolezza delle ambizioni che tali prototipi incarnavano. Sarebbe un vero peccato se proprio oggi, mentre si fa un gran parlare di "informatica pervasiva" (o "*ubiquitous computing*") in quanto aspetti tecnologici particolari dell'informatica sono migrati all'interno di palmari, di dispositivi legati alla casa, all'auto, ecc., l'idea originaria venisse persa di vista a favore di logiche proprietarie e di mercato tendenti a riportare in auge soluzioni *ad hoc* o linguaggi programmativi di nicchia, riducendo di fatto il calcolatore alle funzionalità di una mera calcolatrice cablata, sia pure di tipo sofisticato.

1.4 Il metodo procedurale algoritmico

Come già ricordato il termine informatica deriva da *informatique*, contrazione delle parole francesi *infor*mation auto*matique*. Esse sottintendono l'idea di una elaborazione *automatica* delle informazioni, dove l'automatismo è in qualche misura caratterizzato da un aspetto procedurale della successione di operazioni elementari che vengono applicate all'informazione da elaborare con lo scopo di risolvere un problema del mondo reale. Un tale procedimento parte dunque da un insieme di informazioni fornite dall'esterno al sistema di elaborazione, i cosiddetti *dati iniziali*, che vengono successivamente elaborati secondo una procedura ordinata e a passi, descrivibile in modo preciso ed esauriente come successione finita $I(1), I(2), \ldots, I(n)$ di n *istruzioni elementari*. Il risultato dell'esecuzione dell'istruzione $I(k)$ dipende, oltre che dal tipo di istruzione, anche dai dati d'ingresso allo stesso passo, che sono costituiti tanto da eventuali dati esterni resi disponibili al passo k, quanto dai dati che derivano

dall'esecuzione dell'istruzione $I(k-1)$ al passo precedente. E' evidente che lo svolgimento di una tale elaborazione procedurale, detta anche *computazione*, ha uno scopo preciso, che dal punto di vista oggettivo corrisponde molto spesso alla *risoluzione* di un qualche problema numerico. Il legame col problema del mondo reale viene stabilito grazie al potere simbolico della matematica; in tale contesto ogni simbolo (o numero) può rappresentare un qualunque ente esterno al contesto della computazione, e ciò grazie a un'operazione astratta di *codifica*, che si esplica mediante la definizione di una corrispondenze tra oggetti del mondo interno (simboli o numeri) e oggetti del mondo esterno (enti che a essi corrispondono). La codifica ci consente dunque di trasformare l'enunciazione di un problema del mondo reale in una equivalente di natura simbolica, che va poi risolta mediante il sistema di elaborazione. In ambito informatico, alla procedura in questione viene attribuito il nome di *algoritmo*, la sua esplicitazione rigorosa in un linguaggio comprensibile alla macchina viene chiamata *programma*, mentre si riserva il termine di *computazione* al processo che consiste nell'esecuzione dell'algoritmo (del programma) a partire dai dati iniziali. Il risultato della computazione, cioè i dati generati in uscita al termine della stessa, consente di esprimere in modo codificato la soluzione al problema del mondo reale.



Definizione 1.1. *Dicesi algoritmo una procedura effettiva a passi, descritta in modo preciso ed esauriente da un numero finito di istruzioni elementari $I(1), I(2), \ldots, I(n)$, la quale a partire dai dati iniziali fornisce in un tempo finito i dati finali, che vengono interpretati come la soluzione di un determinato problema. Per computazione s'intende l'esecuzione dell'algoritmo.*

Lo schema che abbiamo testé delineato non è l'unico possibile che consenta di ottenere la soluzione di un problema (il più delle volte di natura matematica) descrivibile secondo un approccio simbolico. Esso è tuttavia quello largamente più usato, e ciò essenzialmente per tre ordini di motivi.
Il primo deriva senza dubbio dal fatto che gli aspetti logico-fondazionali del modello computazionale legato alle procedure effettive furono acquisiti solidamente già nel corso degli anni Trenta, grazie al lavoro dei logici di cui si è accennato precedentemente (Gödel, Church, Turing ecc.). Sulla base di quegli studi si fu in grado di descrivere in termini precisi, cioè in termini matematici, che cosa significhi effettuare una computazione, quali siano i possibili modelli di computazione e quali siano (se esistono) i limiti strutturali all'approccio computazionale. Si riuscì inoltre a stabilire l'equivalenza dei vari modelli proposti, giungendo alla descrizione del modello più generale possibile di elaboratore, la cosiddetta Macchina di Turing (MdT). Essa è in grado di effettuare qualunque computazione, per complessa che possa essere, effettuata da un qualunque elaboratore reale. La definizione della MdT portò inoltre all'enunciazione della celebre tesi di Church-Turing, secondo la quale tutto ciò che si ritiene computabile effettivamente è computabile con una Macchina di Turing. Ricordiamo ancora che le fondamenta teoriche dell'Informatica furono consolidate ben prima che nascessero materialmente i calcolatori, cioè le macchine per eseguire le computazioni in modo efficiente.

Il secondo è legato alla circostanza che tale procedimento è di gran lunga il più immediato e intuitivo. Tutti noi, nella vita quotidiana, adottiamo inconsapevolmente delle strategie di tipo *algoritmico* per risolvere alcuni dei nostri problemi. Per strategia algoritmica intendiamo non solo che essa possa essere esplicitata nei termini precisi di una sequenza di azioni elementari effettive, eseguite le quali si perviene alla soluzione del problema, ma anche che la necessità di dover dare struttura procedurale alla nostra attività ci aiuta a delineare i sotto-problemi che via via siamo chiamati a risolvere, prima di giungere alla soluzione globale. La tecnica algoritmico-procedurale ci offre dunque anche la possibilità di progettare meglio e analizzare con maggior efficacia l'insieme globale delle operazioni elementari da svolgere per conseguire l'obiettivo.
Il terzo ordine di motivi che porta alla popolarità delle procedure algoritmiche è che esse sono state facilmente ed efficacemente cablate nella struttura architetturale delle macchine che devono svolgere materialmente la computazione, cioè degli odierni calcolatori. Vedremo infatti nel seguito che la quasi totalità degli elaboratori oggi impiegati dispone di un'architettura (quella di von Neumann, dal matematico ungherese che per primo la ideò) che si basa sul paradigma di una elaborazione procedurale a passi, di tipo *seriale*, eseguita sequenzialmente lungo la linea temporale da un'unità centrale di elaborazione chiamata *processore*.

Nell'architettura di von Neumann, nota in gergo tecnico anche con la denominazione di *architettura seriale*, il processore esegue una singola istruzione elementare per unità di tempo, e l'intera computazione, anche quando molto complessa, si sviluppa attraverso la concatenazione di un certo numero di passi elementari, basati sul programma che descrive l'algoritmo. La soluzione al problema la si ricava come effetto dell'esecuzione dell'ultima istruzione oppure a seguito di un'uscita forzata dal programma, in quanto la soluzione è stata raggiunta prima.

L'architettura seriale si contrappone alle architetture di tipo *parallelo*, nelle quali la soluzione del problema emerge invece dal comportamento collettivo di un insieme di unità computazionali elementari (per esse si usa talvolta il termine metaforico di *neurone*), il cui comportamento e le cui interazioni non sono descrivibili secondo il paradigma della computazione procedurale. Tali unità computazionali sono connesse in una rete, nella quale ciascun neurone può essere potenzialmente connesso a ciascun altro neurone, dotando così la struttura di un massiccio parallelismo che richiama la topologia usata dalle strutture biologiche nella costituzione del cervello, cioè l'organo che esegue le "computazioni" necessarie all'organismo per la risoluzione dei "problemi" legati alla sua sopravvivenza.
La *computazione parallela* ha avuto dei momenti di grande entusiasmo, che si sono però sempre smorzati a causa della mancanza di un vero e proprio modello matematico generale che fosse in qualche misura la controparte di quello scoperto da Gödel, Church e Turing per l'approccio algoritmico-procedurale. A tutt'oggi non è del tutto chiaro se l'approccio parallelo possa, almeno in linea di principio, collocarsi su un piano essenzialmente diverso rispetto a quello tradizionale, anche se si è ragionevolmente portati a ritenere che la tesi di Church-Turing sia valida anche per le computazioni realizzate secondo una modalità operativa parallela. Da questo punto di vista il passaggio da computazione procedurale a computazione parallela corrisponderebbe, essenzialmente, a una semplice mutuazione della *complessità algoritmica* (cioè del numero di passi elementari necessari per eseguire una determinata computazione, fissati che siano i dati iniziali) con la *complessità strutturale* della rete, che in prima approssimazione si può assumere equivalente alla quantità di memoria necessaria per costruire materialmente la rete stessa.

Anche se finora per le computazioni di tipo esatto (cioè quelle che richiedono una soluzione esatta del problema) l'approccio algoritmico si è dimostrato di gran lunga il più efficiente, non si può escludere che in futuro, anche se solo limitatamente a problemi per i quali ci si accontenti di una soluzione approssimata, diventi più conveniente l'impiego delle cosidette *reti neuronali* (artificiali), oppure di altri paradigmi di computazione che hanno suscitato l'interesse degli esperti, quali la *computazione DNA*, gli *algoritmi genetici* e la *computazione quantistica*. A corroborare tale aspettativa c'è comunque il dato di fatto che la natura, nell'organizzare la struttura dei suoi organismi biologici, "ha scelto" il modello parallelo, che gode di una serie di vantaggi derivanti dal fatto che la computazione è sostanzialmente innervata nella struttura della rete. In questo testo ci occuperemo tuttavia del solo approccio algoritmico-procedurale.

In conclusione vale la pena riflettere ancora sulla circostanza, già anticipata nella prefazione, che l'Informatica intesa come studio delle procedure effettive (algoritmi) prescinde dalla tecnologia che s'impiega per l'attuazione delle procedure stesse, al punto che è concepibile, in linea di principio, un'informatica senza calcolatori. La prova di ciò risiede nella comprensione del modello computazionale astratto della macchina di Turing (o di qualunque altro ad esso equivalente) che consente di eseguire una procedura algoritmica facendo uso solo di carta

e penna. Poiché sulla base della tesi di Church-Turing tutto ciò che è computabile nel senso comune e intuitivo del termine lo è anche nel senso della Macchina di Turing, si deduce che tutto ciò che siamo in grado di attuare coi nostri moderni calcolatori sarebbe in linea di principio realizzabile in modo effettivo anche solo con carta e penna. Pur tuttavia gli straordinari successi dell'informatica sono stati resi possibili solo da uno sviluppo impetuoso della tecnologia microelettronica, che rappresenta oggi il sostrato fisico sul quale viene cablata l'architettura del calcolatore. Da questo punto di vista il vero salto di qualità lo si ebbe alla fine degli anni '40, quando fu inventato il *transistor*, che diede il via al processo di miniaturizzazione che consente oggi di disporre, sul palmo di una mano, della potenza di calcolo equivalente a quella di grossi elaboratori che, prima di tale invenzione, occupavano intere ali di edifici consumando decine di kilowatt. Di questa *rivoluzione microelettronica* daremo conto nella prossima sezione.

1.5 La rivoluzione microelettronica e la legge di Moore

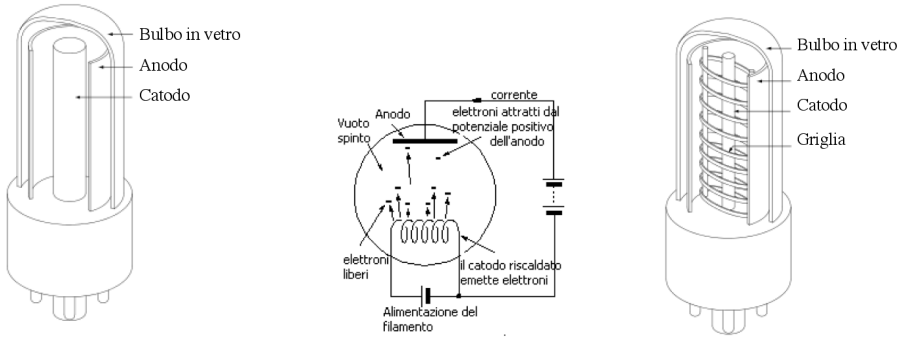
L'Informatica odierna è una tecnologia pervasiva, che sta modificando non solo le nostre capacità di elaborazione e di risoluzione di problemi complessi, ma persino le nostre relazioni interpersonali. La possibilità di integrare le capacità di elaborazione dei moderni computer con la connettività offerta da *Internet* sta facendo emergere nuovi paradigmi di partecipazione e di interazione sociale (si pensi al fenomeno dei cosiddetti *Social Network*), che portano profonde modifiche in seno alla società.
Anche se siamo ormai abituati all'idea di stabilire relazioni con altre persone, consultare pagine web o fare fotografie con uno *smartphone* che si porta agevolmente in un taschino, bisogna rendersi conto che tutto ciò non è un prodotto scontato dell'evoluzione tecnico-scientifica. L'acquisizione di un solido modello di computazione, introdotto nei primi anni '30 secondo il processo storico delineato nella sezione 1.3 e la costruzione dei primi calcolatori basati sul modello architetturale di von Neumann, non avrebbero in ogni caso consentito lo sviluppo dell'Informatica che oggi conosciamo se, contemporaneamente, non si fosse realizzata una vera e propria *rivoluzione microelettronica*, che partendo dai primi tubi a vuoto (i diodi e i triodi di figura 1.7), che erano in grado di dare consistenza circuitale alle variabili Booleane necessarie per la costruzione di un computer, ha portato alla costruzione di circuiti integrati ad altissima scala di integrazione (VLSI - *Very Large Scale Integration*), che comprendono milioni o anche miliardi di dispositivi logici equivalenti, dal punto di vista della funzione, ai citati tubi. Diamo allora un breve cenno sull'evoluzione della tecnologia elettronica per comprendere come, dai primi dispositivi a vuoto, si sia giunti agli straordinari *chip* che corredano i nostri computer.

(a) Alcuni tipi di tubi a vuoto o termoionici

(b) Tubo con involucro *subminiatura*

Figura 1.13: Evoluzione dei tubi termoionici, dai primi modelli con zoccolo in bakelite (a), alle ultime realizzazioni degli anni '80 (b), denominate *subminiatura*

Subito dopo il raggiungimento di un'affidabilità adeguata dei tubi a vuoto, che si ottiene solo a partire dalla metà degli anni '20, inizia una produzione industriale su vasta scala, sostenuta inizialmente dalla spinta della tra-



(a) Struttura interna di un diodo a vuoto

(b) Principio di funzionamento del diodo

(c) Struttura interna del triodo; si noti l'elettrodo di controllo, denominato *griglia*

Figura 1.14: Struttura e principio di funzionamento dei tubi a vuoto

sformazione delle prime radio a *galena*, completamente prive di componenti elettronici *attivi*, in radioricevitori ad *amplificazione diretta*, nei quali i tubi termoionici vengono usati per *amplificare* i deboli segnali ricevuti dall'antenna. Lo sviluppo tecnologico porta a una diminuzione dei volumi d'ingombro (si veda la figura 1.13a) e a una integrazione dello zoccolo (inizialmente realizzato in *bakelite*) nel bulbo di vetro. Il culmine della tecnologia dei tubi a vuoto si raggiunge nei primi anni '80, durante i quali vengono realizzati i tubi *subminiatura*, che avevano una lunghezza di circa 3 cm. Il funzionamento dei tubi termoionici è illustrato in figura 1.14. Il tubo è costituito da un bulbo di vetro nel quale viene praticato il vuoto. Un *filamento*, simile a quello di una comune lampadina a incandescenza, riscalda una superficie cilindrica chiamata *catodo* (3.3a), ricoperto di ossidi speciali; l'energia associata al riscaldamento libera un certo numero di elettroni del catodo, che formano una nuvola intorno ad esso; poiché gli elettroni hanno una carica elettrica negativa, essi possono essere attirati da un elettrodo polarizzato positivamente e denominato *anodo*. Questo fenomeno determina un flusso di elettroni che va dal catodo all'anodo, cioè una corrente unidirezionale (fig. 3.3b). Se tra anodo e catodo si interpone un terzo elettrodo, denominato *griglia* (3.6a), il flusso di corrente può essere comandato dal potenziale (negativo) con il quale viene caricata la griglia. Ciò significa che una piccola variazione del potenziale della griglia, che è molto vicina al catodo, può determinare una grossa variazione della corrente anodica, e da ciò nasce l'attitudine del tubo termoionico ad *amplificare* i segnali.

Il principio fisico che sostiene il funzionamento dei tubi a vuoto è anche il principale responsabile dell'impossibilità di scendere, con la miniaturizzazione, al di sotto di certe dimensioni minime; infatti i potenziali anodo-catodo necessari per un corretto funzionamento di un generico tubo a vuoto sono tipicamente dell'ordine delle centinaia di Volt, e non possono scendere in ogni caso al di sotto dei 50-60 V, che rappresenta la minima tensione usata per i tubi subminiatura. Di conseguenza un'ipotetica ulteriore diminuzione dei volumi e delle distanze interelettrodiche comporterebbe delle scariche disruptive tra gli elettrodi, poichè verrebbe superata la tensione di isolamento tra gli stessi. La spinta verso la miniaturizzazione dei tubi a vuoto si fermò dunque ai tubi subminiatura, che non avrebbero consentito neanche la costruzione di calcolatori da tavolo.

Tuttavia già dai primi anni '50 iniziava a farsi strada una nuova tecnologia elettronica, quella dei *semiconduttori*. Nel 1947 *William Shockley, John Bardeen* e *Walter Brattain*, dei laboratori *Bell*, realizzarono il primo prototipo funzionante di *transistor* (che era del tipo *a punta di contatto*, vedi fig.1.15), costituito da un cristallino di *germanio* e da tre elettrodi. Il transistor realizza la stessa funzione dei tubi termoionici, cioè quella di poter controllare, con un elettrodo intermedio denominato *base*, un flusso di corrente che scorre tra altri due elettrodi, l'*emettitore* e il *collettore*. La differenza sta però tutta tra le dimensioni dei due componenti, poiché il cuore del transistor, nella più moderna versione *a giunzione*, è costituito da un microscopico cristallino di germanio o silicio quasi privo di spessore e di meno di un millimetro quadrato, mentre il volume di un tubo è di diversi cm cubici. Il punto di partenza per la costruzione di un transistor è l'impiego della citata *giunzione* PN; essa è costituita da un cristallino di silicio nel quale vengono inseriti, tramite un drogaggio chimico, atomi di elementi diversi, realizzando in tal modo una sezione P e una sezione N. Il silicio ha 4 elettroni nell'orbita esterna; se ad esso vengono aggiunti atomi di un elemento chimico con tre elettroni nell'orbita esterna (p.es. il *Boro* o l'*Indio*) si ottiene un cristallo

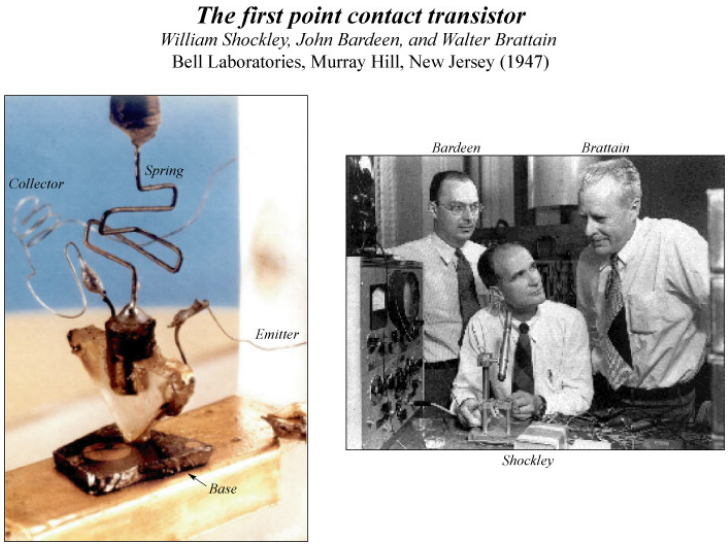


Figura 1.15: Il primo transistor realizzato nei laboratori della Bell da Shockley, Bardeen e Brattain

drogato P. Viceversa, se aggiungiamo atomi di un elemento chimico con cinque elettroni nell'orbita esterna (p.es. l'*Arsenico* o il *Fosforo*) si ottiene un cristallo drogato N. In corrispondenza di ogni atomo dell'elemento drogante di tipo P, poiché ci sono solo tre elettroni nell'orbita esterna, resta disponibile uno spazio vuoto o *lacuna*, che può essere riempita dagli elettroni in eccedenza della giunzione N (vedi fig.1.16a, dove le lacune sono in giallo e gli elettroni in rosso). Se si fornisce una tensione positiva al lato P della giunzione, si instaura un flusso di elettroni da N a P (fig.1.16c), mentre se si inverte la polarità il flusso di corrente si blocca (fig.1.16d). Una giunzione PN ha quindi la funzione di un *diodo*, poiché consente un flusso solo unidirezionale della corrente.

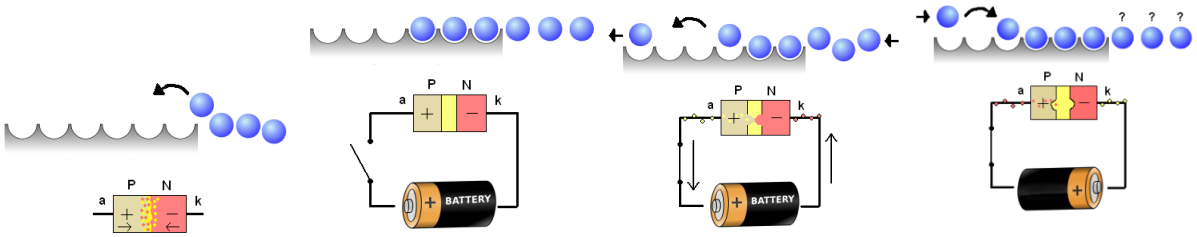
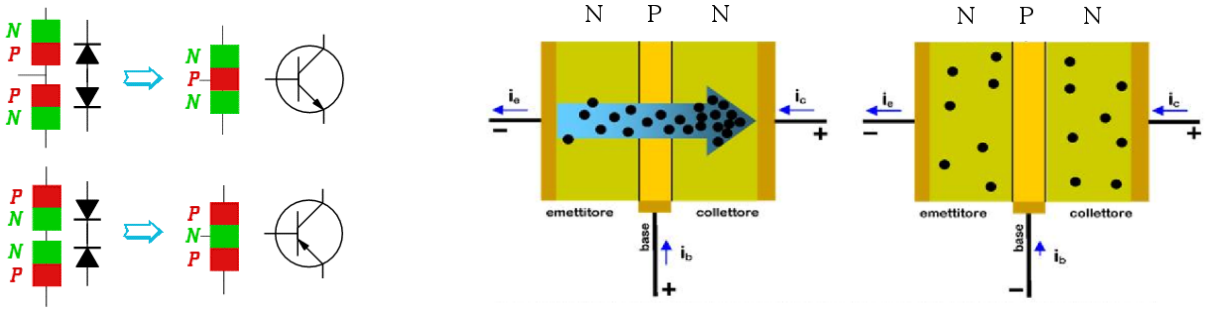


Figura 1.16: Principio di funzionamento della giunzione PN

Il transistor viene realizzato accoppiando due giunzioni PN nei due modi possibili, cioè NPN e PNP (fig. 1.17a). Il funzionamento del dispositivo è brevemente illustrato dalla figura 1.17b. Quando la base è polarizzata positivamente rispetto all'emettitore (negativamente per il caso PNP), la giunzione conduce e una percentuale rilevante di elettroni attraversa la base, che è molto sottile, e perviene al collettore. Se la polarizzazione viene invertita il flusso di corrente si blocca. Il guadagno che ne deriva, dal punto di vista dei volumi occupati dai due dispositivi, è ben evidente in figura 1.18a. Ad esso si aggiunge anche una maggior robustezza rispetto agli urti e la possiilità di impiegare basse tensioni (pochi Volt) al posto delle tipiche tensioni in uso per i tubi, che come detto erano superiori ai 100V.
La tecnologia dei transistor si sviluppa molto rapidamente, e la tecnica *a punta di contatto*, con la quale venne realizzato il primo prototipo, venne subito abbandonata dallo stesso Shockley, che l'anno successivo ideò il transistor a giunzione appena descritto. I primi dispositivi prodotti in larga scala erano realizzati con cristalli di



(a) Transistor a giunzione NPN e PNP

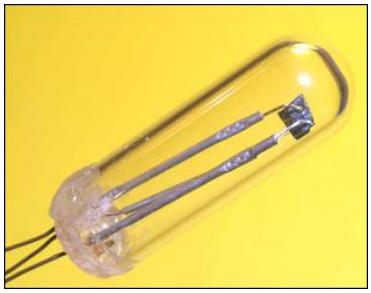
(b) Principio di funzionamento del transistor

Figura 1.17: Transistor e suo principio di funzionamento

germanio, incapsulati in un contenitore di vetro (verniciato in nero) o di metallo per le tipologie che necessitavano un migliore smaltimento del calore (fig.1.18b). La figura (fig.1.18c) ci consente di evidenziare il fatto che quasi tutto l'ingombro, pur minimo, del transistor, serve per poter consentire il collegamento del cristallo con l'ambiente esterno, tramite dei reofori che devono garantire un'adeguata resistenza alle sollecitazioni.

(a) Confronto tra le dimensioni di un tubo termoionico degli anni '80 e di un transistor al germanio per basse potenze

(b) I primi transistor al germanio prodotti su larga scala negli anni '70

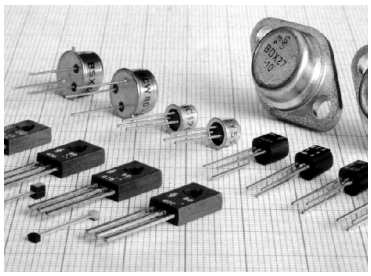


(c) Particolare della giunzione di un transistor al germanio

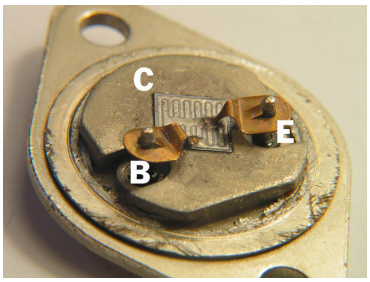
Figura 1.18: I primi transistor prodotti in larga scala negli anni '70 a confronto con la tecnologia dei tubi a vuoto, oramai al tramonto

Il germanio viene presto sostituito col silicio per motivi di affidabilità, in quanto quest'ultimo è molto più resistente alle temperature ($180-200\,^{\circ}$C contro $80-100\,^{\circ}$C) e con una minore deriva termica (aumento della corrente con la temperatura della giunzione). La tecnologia si consolida rapidamente, sviluppando una molteplicità di dispositivi adatti a tutte le applicazioni (fig. 1.19a), anche quando queste prevedono una dissipazione intensa di potenza, per la quale si progettano dei contenitori molto robusti in metallo, per poter connettere il dispositivo a dei dissipatori mediante l'uso di viti (fig. 1.19b).

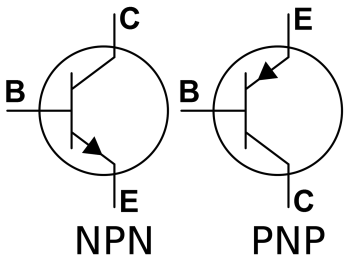
La possibilità di drogare un microscopico cristallino di silicio in modo P e N per realizzare un transistor fa quasi subito prospettare l'idea che su uno stesso sostrato di silicio, denominato *wafer*, fosse in linea di principio possibile realizzare più transistor connessi circuitalmente assieme. E in effetti nel 1958 *Jack Kilby* realizza alla *Texas Instrument* il primo *circuito integrato* della storia, che contiene due transistor connessi su una barretta di germanio (fig.1.20a). Due anni dopo viene annunciato il primo circuito integrato prodotto commercialmente, il multivibratore TI 502 (fig. 1.20b), che contiene 2 transistor, due diodi, 6 resistenze e 4 condensatori (fig. 1.20c) e che costa oltre 500 $. Nel 1965 la stessa *Texas Instrument* realizza un amplificatore integrato di circa 1,5 mm^2, con 7 transistor, che viene prodotto su larga scala con un significativo abbattimento dei costi (fig.1.21a). E' iniziata l'era dell'integrazione su larga scala.



(a) Alcuni esempi di transistor in plastica e metallo

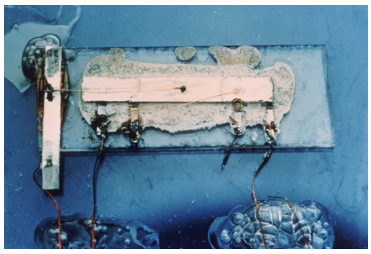


(b) Interno di un transistor di potenza

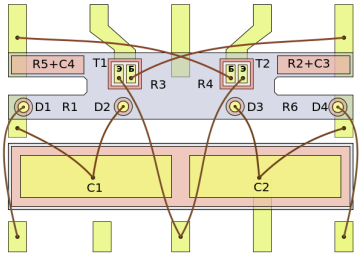


(c) Simbolo elettrico dei transistor NPN e PNP

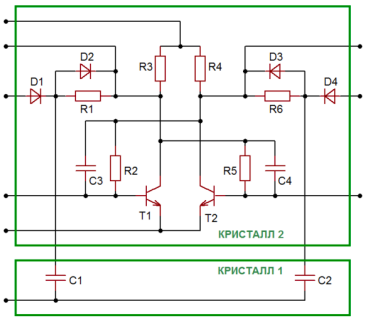
Figura 1.19: Alcuni tipi di transistor e i simboli elettrici relativi



(a) Il primo circuito integrato, realizzato da Jack Kilby nei laboratori della *Texas Instrument*



(b) Struttura interna del multivibratore TI 502

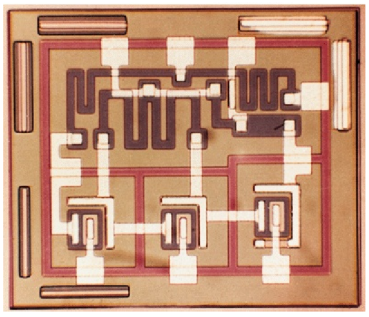


(c) Schema del multivibratore TI 502

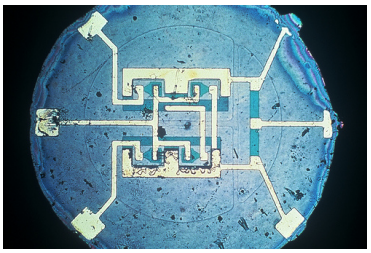
Figura 1.20: Primi circuiti integrati prodotti dalla *Texas Instrument*

Dal punto di vista informatico l'aspetto più rilevante è legato alla possibilità di poter integrare tutti i circuiti necessari per poter realizzare le *porte logiche booleane*, cui si è fatto cenno nella figura 1.4 e che costituiscono il nucleo della circuiteria dei computer. Come esempio si può vedere in figura (1.22) la tabella dei valori di verità per una porta NOR, la sua realizzazione circuitale e il circuito integrato corrispondente. Il funzionamento della porta NOR è il seguente: se entrambi gli ingressi sono posti al valore 0 (False) allora l'uscita è al valore 1 (True); se invece anche uno solo o entrambi hanno valore 1, allora l'uscita va al valore 0. I valori logici 0 e 1 sono realizzati circuitalmente distinguendo tra tensione bassa (0 V) e tensione alta (p.es. 5 V). La linea di sviluppo dei circuiti integrati segue due direttive indipendenti. La prima è quella dei circuiti che amplificano o manipolano un segnale *analogico*, cioè un segnale continuo che si sviluppa nel tempo (p.es. un segnale sonoro nel caso di un amplificatore audio). La seconda è quella delle *porte logiche*, nella quale su ciascun involucro (o *case*) vengono integrate un certo numero di porte AND, OR, NOT, NAND, XOR (vedi fig.1.23). Con questi circuiti logici, che appartengono alla grande famiglie degli integrati TTL (*Transistor-Transistor-Logic*) introdotti sul mercato dalla *Sylvania* nel 1963 e dalla *Texas Instrument* nel 1964, si costruiscono i primi computer personali, quali ad esempio il *Kenbak-1* (fig. 1.23c), del 1971, fino all'Olivetti 6060, che nel 1975 fu uno degli ultimi a usare le porte TTL come elementi costitutivi dell'*Unità Centrale* (o *Central Processing Unit*) del computer.

Nel frattempo, nei laboratori della *Intel*, il progettista capo *Federico Faggin* porta a compimento la realizzazione del primo *microprocessore*, il 4004 (si veda fig.1.24a e 1.24b), che è un circuito integrato di 2300 transistors contenente tutti i principali blocchi logici della CPU in un unico *case*. L'innovazione è notevole, poiché consente di abbattere ulteriormente i costi di produzione dei computer, rendendoli nel contempo meno ingombranti e più affidabili. Al 4004 segue l'anno successivo il modello 8008 (fig.1.24c e 1.24d), che conta ben 3500 transistors. La *Intel* diventa azienda leader nel settore della produzione di microprocessori, e uno dei suoi co-fondatori, *Gordon E. Moore*, osserva in un suo articolo del 1965 che l'evoluzione dei circuiti integrati è stata così veloce da consentire un



(a) Ingrandimento del primo circuito integrato prodotto a basso costo su larga scala



(b) Connessione con l'esterno di un circuito integrato a 6 piedini

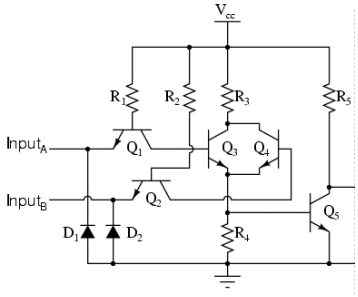
(c) Struttura interna di un circuito integrato per montaggio superficiale

Figura 1.21: Alcuni esempi di struttura interna di circuiti integrati commerciali

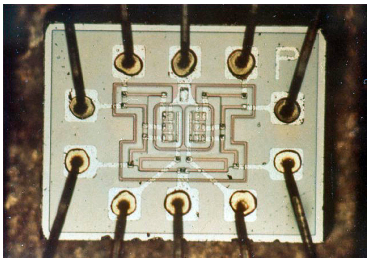
Truth Table

Input A	Input B	Output Q
0	0	1
0	1	0
1	0	0
1	1	0

(a) Tabella di verità della funzione NOR



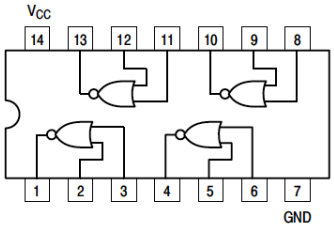
(b) Schema circuitale corrispondente



(c) Circuito integrato che la realizza circuitalmente

Figura 1.22: Circuito integrato che realizza una funzione NOR

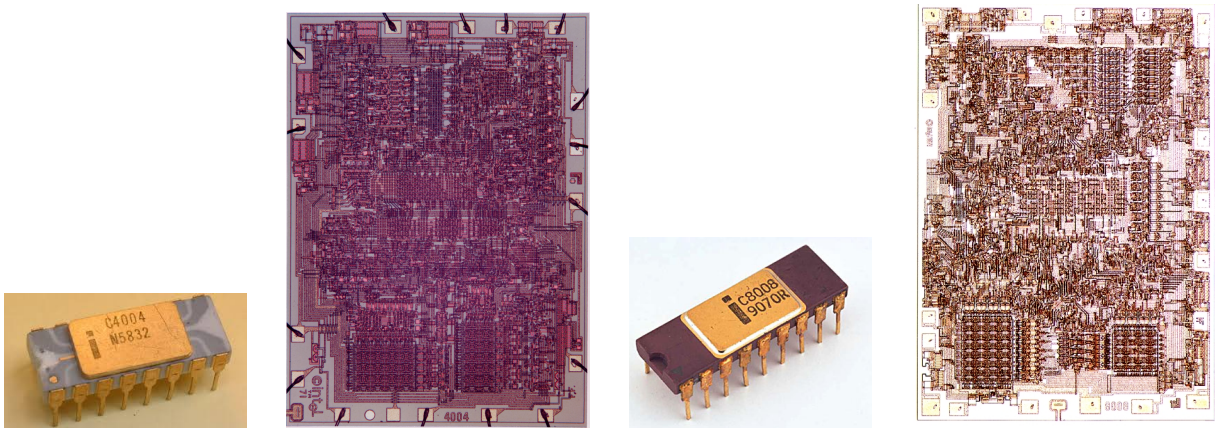
(a) Contenitore per un circuito integrato con quattro porte logiche di tipo NOR



(b) Connessione interna delle porte

(c) Uno dei primi *personal computer*, il *Kenbak*-1

Figura 1.23: La famiglia di circuiti integrati TTL e uno dei primi *computer* che fece uso di questa tecnologia



(a) *Intel* 4004 (1971) (b) 4004 con 2300 transistor (c) *Intel* 8008 (1972) (d) 8008 con 3500 transistor

Figura 1.24: *Case* e circuito integrato dei processori 4004 e 8008 della *Intel*

raddoppio del numero di componenti attivi (transistor) ogni (circa) due anni dal 1958, anno della costruzione del primo integrato, al 1965. Moore azzarda anche una previsione, e cioè che una tale legge esponenziale di crescita sarebbe durata "almeno per una decina d'anni". Oggi son passati quasi cinquant'anni da quella previsione, e la sua validità non è stata minimamente scalfita dal passare del tempo. Nella figura 1.26 si possono vedere le più importanti famiglie di processori prodotte fino a oggi, e osservando il numero di transistor contenuti in ciascun processore ci si rende conto che l'incremento è stato effettivamente esponenziale; se il 4004 aveva 2300 transistor, i più recenti processori a 8 *core*, cioè con 8 processori fisici montati su un unico contenitore e che lavorano in modo coordinato, si arriva al numero astronomico di 2270 milioni di transistor! Recentemente, nel maggio 2020, la Nvidia ha presentato il suo ultimo processore A100 AI (si veda la figura 1.25), con 54 miliardi di transistors e accreditato di una performance di 5 *petaflops*, cioè $5 \cdot 10^{15}$ *FLoating point Operations Per Second*.

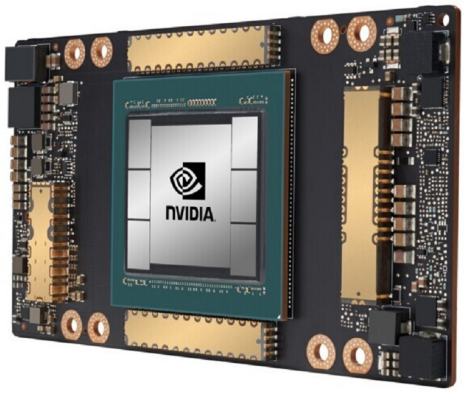
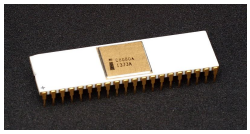


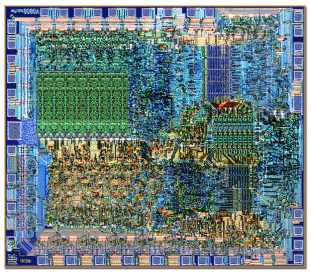
Figura 1.25: Processore A100 AI Nvidia, con 54 miliardi di transistors e 5 petaflops

La previsione nella crescita della scala di integrazione fu talmente precisa che alla stessa fu attribuita, successivamente, la qualifica di *legge di Moore*; si osservi che a tutt'oggi tutte le aziende del settore la usano come elemento di previsione per poter pianificare i progetti futuri e stabilire gli obiettivi della miniaturizzazione. Il diagramma di figura 1.27 ci mostra in ordinata il numero di transistor (in scala logaritmica) mentre nel diagramma sono riportate le varie famiglie di processori che si sono succeduti nella produzione *Intel*. Si può notare che la curva di crescita è effettivamente esponenziale (una retta su scala logaritmica).
I circuiti integrati hanno subìto un abbattimento dei costi e un incremento di prestazioni che non hanno eguali in

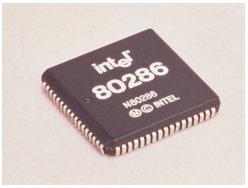
tutta la storia della tecnologia, poichè il raddoppio dei componenti ogni 24 mesi comporta di fatto un raddoppio delle prestazioni ogni 18 mesi circa. A tutt'oggi la crescita esplosiva garantita dalla legge di Moore non sembra dar segni di cedimento, anche se le dimensioni dei componenti e delle connessioni tra essi cominciano ad essere paragonabili con le dimensioni delle strutture atomiche. Come vedremo nel capitolo relativo agli aspetti architetturali, la crescita esponenziale nel numero di transistor porta con se una serie di conseguenze positive un po' su tutta l'architettura dei computer, a cominciare dalle memorie, che implica un incremento relativo delle prestazioni ancora maggiore. di quanto ci si aspetti.



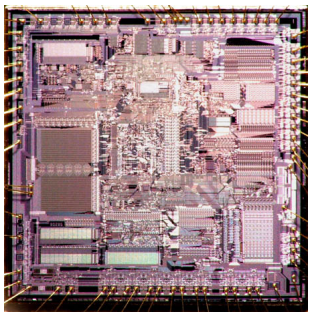
(a) *Intel* 8080 (1974)



(b) 8080 con 4500 transistor

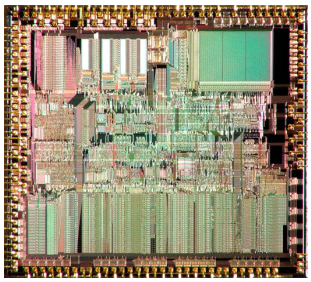


(c) *Intel* 80286 (1982)



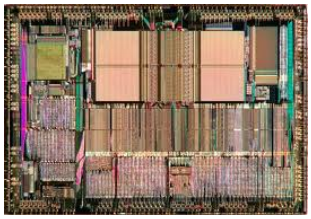
(d) 80286 con 134000 transistor

(e) *Intel* 80386 (1985)



(f) 80386 con 275000 transistor

(g) *Intel* 80486 (1989)

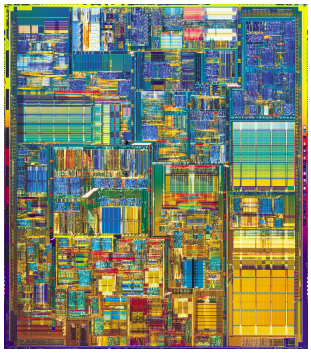


(h) 80486 con 1,2 M di transistor

(i) *Intel* Pentium (1993)

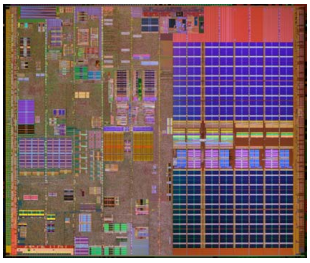
(j) Pentium con 3,1 M di transistor

(k) *Intel* Pentium 3 (1999)



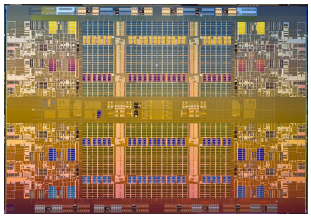
(l) Pentium 3 con 9,9 M di transistor

(m) *Intel* Pentium 4 (2004)



(n) Pentium 4 con 125 M di transistor

(o) *Intel* Xeon 8 *core* (2010)



(p) Xeon con 2270 M di transistor

Figura 1.26: Sviluppo temporale dei principali processori della *Intel*

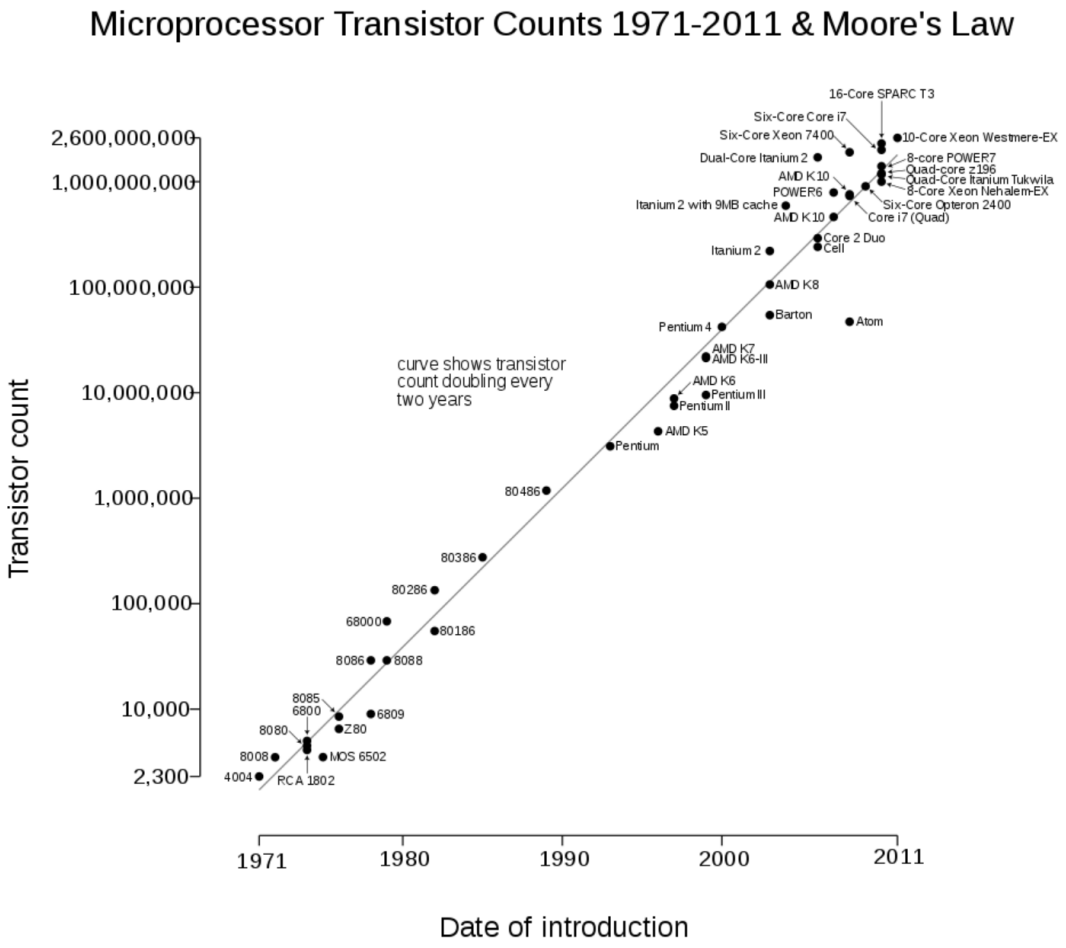


Figura 1.27: Diagramma che illustra la validità della legge di Moore

Capitolo 2

Informazione e mondo esterno

Abbiamo già ricordato che in molti casi un problema del mondo esterno può essere risolto descrivendolo mediante un'opportuna codifica astratta, sulla quale si applica nel seguito una procedura algoritmica di risoluzione. Se la codifica e la procedura sono realizzate correttamente, il risultato simbolico ottenuto dopo un tempo finito può essere decodificato nei termini di una soluzione al problema primitivo. Anche se i problemi che la macchina risolve sono di natura strettamente logico-matematica, la potenza espressiva della codifica simbolica consente di gestire problemi del mondo reale che possono spaziare fra tipologie molto diverse tra loro. Come esempio si possono confrontare il problema di manipolare alcune immagini digitali, per esempio per effettuare un foto-ritocco, con quello di trovare il percorso ottimale, tenendo conto dei sensi unici, che un furgone deve effettuare per approvvigionare i diversi supermercati di una grande città. O ancora il problema dell'individuazione di una o più parole chiave all'interno delle miliardi di pagine disponibili nella rete di Internet, tramite il cosiddetto *motore di ricerca*, con quello del calcolo della traiettoria ottimale per il lancio di un vettore spaziale. Come ultimo esempio, quello forse più estremo, citiamo la ricostruzione, fatta dalla macchina, di un ambiente virtuale come l'interno di una piramide, che può essere "visitato" comodamente stando seduti su una poltrona, come se la visita fosse fatta in prima persona. Da un punto di vista sistemico la situazione è allora quella descritta nella figura 2.1, dove viene messo in evidenza un ambiente esterno, quello del mondo reale, in cui le grandezze fisiche (o dati d'ingresso), associate implicitamente all'enunciazione e alla risoluzione di un certo problema, devono essere codificate in simboli comprensibili al calcolatore. I dati d'ingresso possono essere costituiti da lettere su un alfabeto discreto, tipicamente i simboli di una tastiera di un calcolatore o dalle cifre necessarie a comporre i numeri che si usano per fare i calcoli, ma anche da immagini o da da suoni o più in generale da segnali che possono derivare da strumenti di misura ecc. Come vedremo fra poco i calcolatori lavorano sulla base di un alfabeto binario, e ciò per tutta una serie di motivi che illustreremo nel seguito. Tutte queste informazioni devono dunque essere trasformate, tramite una *codifica*, in lunghe sequenze di *stringhe binarie*, l'unica tipologia d'informazione che i circuiti elettronici, basati sul sistema della logica Booleana, sono in grado di gestire.

Il calcolatore elabora, secondo l'algoritmo escogitato per risolvere il problema, i dati codificati d'ingresso, fornendo alla fine della procedura un risultato simbolico binario, che va poi decodificato nelle grandezze del mondo esterno che rappresentano la soluzione, in modo da poter essere lette e interpretate da esseri umani.

Dunque, qualunque sia l'algoritmo e comunque funzioni il sistema di elaborazione, il primo passo è quello di studiare il problema della codifica. Per farlo è necessario specificare la tipologia e le modalità di rappresentazione delle grandezze esterne da una parte, e considerare la tipologia di simboli che la macchina è in grado di gestire dall'altra. Cominciamo allora a descrivere le modalità di rappresentazione dei dati d'ingresso.

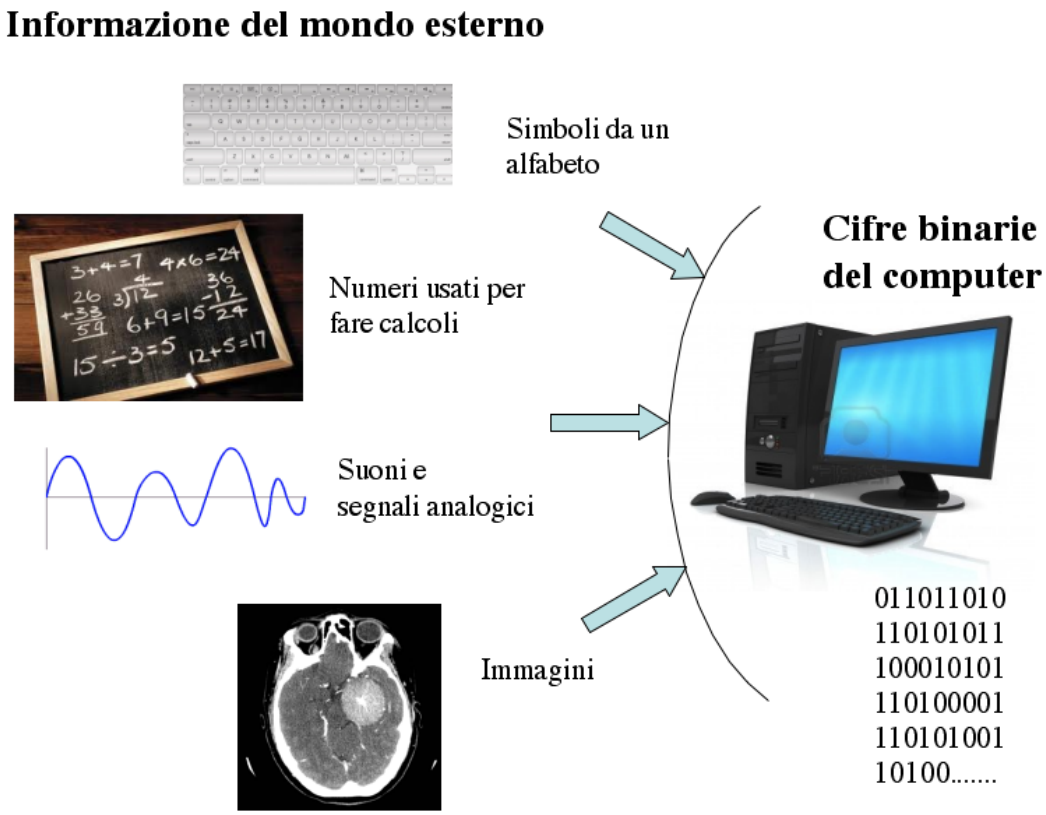


Figura 2.1: Tutte le informazioni del mondo esterno vanno codificate in lunghe stringhe binarie

2.1 Informazione e ridondanza

Quando acquisiamo un dato, la lettera di un alfabeto, un'immagine, un suono, un numero, ecc., stiamo acquisendo *informazione* dal mondo esterno. Se riflettiamo per un momento su quella che è la matrice primitiva dell'informazione, ci rendiamo conto che la sua generazione è in qualche misura associata alla variazione di una qualche grandezza fisica: la lettura di un testo scritto è possibile grazie alla nostra capacità di discriminare la variazione di luminosità tra le lettere nere che compongono il brano e il bianco della pagina; l'ascolto di una frase nel linguaggio parlato è reso possibile dalla differenza di pressione dell'aria generata, tramite le corde vocali, nella bocca del parlante; la percezione di un particolare ritenuto interessante in un immagine, p.es. una macchia chiara in una tomografia, è resa possibile dalla capacità del nostro occhio di valutare differenze di contrasto, e così via.

L'informazione nasce dunque come stato oggettivo per l'osservatore, nel momento in cui questi individua una *differenza* nella grandezza di riferimento, in corrispondenza di due istanti successivi o di due punti dello spazio. Inoltre la quantità d'informazione fornita nell'esperimento è intuitivamente legata alla numerosità degli stati possibili *a priori*. Una volta generata, l'informazione si propaga nell'ambiente esterno alla sorgente attraverso opportuni *canali* di trasmissione, che costituiscono il sostrato fisico attraverso il quale le differenze di cui prima si parlava si propagano al di fuori della sorgente. In generale il canale potrebbe essere un qualunque elemento fisico che consenta la lettura a distanza della configurazione assunta dalla sorgente, ma in pratica, perlomeno nel contesto tecnico-ingegneristico in cui di fatto si finisce con l'operare, esso è quasi sempre una linea di trasmissione (doppino telefonico, cavo coassiale, guida d'onda o fibra ottica), scelta sulla base della tecnologia in auge, e che consenta la trasmissione a distanza dei cosiddetti *segnali* che incorporano l'informazione che la sorgente genera. Si noti che la trasmissione può essere a distanza nello *spazio*, nel qual caso si connettono due punti distanti fisicamente, ma

anche nel *tempo*, quando si effettua una memorizzazione dell'informazione su un opportuno supporto.

L'informazione non ha però una valenza assoluta, poiché dipende in sostanza dalla capacità di discriminazione dell'osservatore a essa interessato; questi ha di fronte a sé un sistema fisico, la *sorgente*, e una (o più) grandezze che lo descrivono, suscettibili di assumere *più stati diversi*. L'informazione rilevata da un secondo osservatore, più acuto del primo e che sia in grado di discriminare tra più stati diversi della stessa grandezza (o tra più grandezze), rimane per il primo osservatore in uno stato di latenza, che si desta solo nel momento in cui anche il primo osservatore dispone della capacità fisico-sensoriale (e della volontà) di rilevare le differenze di cui prima era inconsapevole. Come esempio si può pensare all'esperimento "lancio di un dado", con un osservatore O_1 (che potrebbe essere un giocatore d'azzardo) interessato solamente al numero uscito, e un osservatore O_2 (p.es. uno studente di Fisica) che apprezza anche l'orientamento delle facce, la posizione occupata dal dado nell'istante finale ecc. I due punti di vista sull'informazione associata all'esperimento sono evidentemente molto diversi, e riflettono il diverso tipo di interesse manifestato dai due osservatori nei confronti dell'evento in oggetto. Si noti per altro che lo stato di latenza cui si è accennato precedentemente può derivare anche dall'indisponibilità, da parte dell'osservatore, dei rilevatori sensoriali idonei a cogliere la grandezza fisica associata al funzionamento della sorgente: l'accensione di una luce emessa da una lampada che lavora sull'infrarosso non potrà essere percepità da chi non disponga degli apposti occhiali per la visione notturna.

Le *differenze* di cui si parla hanno una loro strutturazione gerarchica, poiché può *cambiare* il modo in cui si manifesta una differenza; anche questa differenza del secondo livello (differenza di differenze) è rilevabile in modo oggettivo ed è essa stessa fonte d'informazione. Si osservi il segnale rappresentato in figura 2.2(a), che potrebbe essere relativo alla luminosità di una torcia, con la quale si lanciano dei segnali luminosi a distanza, di notte. Il passaggio tra torcia accesa e torcia spenta costituisce una variazione, e quindi un'informazione a livello base dei dati. Tuttavia una variazione nella frequenza delle accensioni/spegnimenti (fig.2.2(b) o (c)), è una variazione del secondo livello, poiché varia la modalità di variazione della luminosità. Riprendendo l'esempio del dado si può invece pensare a una successione di lanci, fatta dallo studente di Fisica, in cui ciascuna faccia abbia una frequenza relativa di (circa) 1/6, seguita da una seconda successione "sospetta", eseguita dal giocatore d'azzardo, in cui il 6 esca con una frequenza stranamente elevata. Se dal punto di vista delle differenze del prim'ordine ci si può limitare a registrare l'accaduto, un'analisi dell'informazione associata alle differenze del second'ordine, cioè al diverso "comportamento" del dado lanciato dal giocatore, porterebbe a un'informazione di secondo livello su una presunta manomissione del dado da parte del giocatore d'azzardo. Questa gerarchia si colloca comunque su un piano

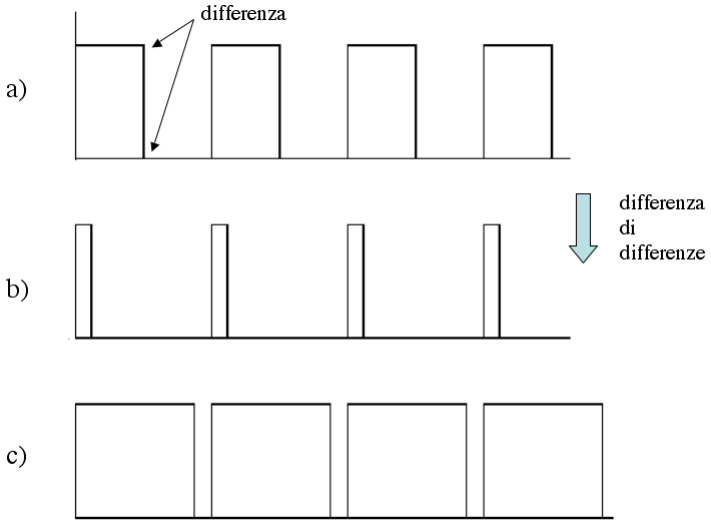


Figura 2.2: L'informazione di primo livello (la variazione del segnale) e quella di secondo livello (la variazione della variazione), che corrisponde in pratica a un cambio nella frequenza del segnale

strettamente *sintattico*, poiché riguarda direttamente i dati. Nell'esempio precedente l'informazione sintattica è

relativa al fatto che il comportamento del dado è cambiato; tuttavia la conseguente congettura sul dado truccato deriva da un'*interpretazione* delle differenze relativamente a un certo contesto logico, l'unico che possa esprimere un *significato* per esse. Ciò induce una gestione delle differenze anche su un piano *semantico*, molto difficile da gestire con un modello matematico formale.

L'informazione ha dunque un carattere relativo, dipende sensibilmente dall'utente che interroga la sorgente, dalla sua capacità di discriminazione, e presuppone una fase *sintattica* o di *rilevamento* della stessa, che si distingue nettamente dalla fase *semantica* relativa alla *comprensione* del significato, processo che consiste essenzialmente nel riportare ad una propria esperienza pregressa i caratteri sintattici emersi dall'informazione disponibile, riconducendo la massa disaggregata di dati verso quel piano di coerenza semantica che porta al *significato*. Come esempio illuminante si rifletta sulla lettura di una frase scritta in una lingua che non si conosce, come *Fartyget kommer att segla på16*, contrapposta alla lettura della stessa frase scritta nella propria lingua, cioè *La nave salperà alle ore 16*. In entrambi i casi avviene il rilevamento, ma la comprensione del significato della frase si ha solamente nel caso in cui l'osservatore conosca la lingua in questione.

Tutto quanto descritto finora ha come immediata conseguenza un'intrinseca difficoltà nel giungere a una quantificazione e a una gestione globale dell'informazione, che tenga cioè conto tanto dei suoi aspetti sintattici quanto di quelli semantici, che possono fra l'altro essere assai poco correlati tra loro.

Il modello di gestione e di trasmissione dell'informazione suggerito da Shannon, nel suo lavoro precedentemente citato, fa riferimento al solo piano sintattico, e ciò costituisce contemporaneamente un pregio e una limitazione. La limitazione è conseguente alla circostanza che, essendo svincolato da connotazioni di tipo semantico, il modello descrive solo in minima parte la ricchezza del processo di comunicazione, così come avviene nel mondo reale. Il pregio risiede nel fatto che tale limitazione, circostanziando la validità del modello, ha consentito di ottenere risultati rilevanti sul piano tecnico-operativo, che hanno dato un impulso straordinario allo sviluppo dell'informatica. Bisogna ricordare che ci sono stati e ci sono tuttora diversi tentativi di inserire la semantica nelle macchine o nei programmi (si ricordi la cosiddetta *Intelligenza Artificiale* o i *Sistemi Esperti*), ma finora i risultati sono stati piuttosto modesti. La semantica presuppone infatti la *conoscenza*, che attraverso l'apprendimento porta all'*esperienza*, dalle cui stratificazioni emerge il profilo semantico; ma questo processo richiede l'acquisizione di una quantità enorme di dati e, soprattutto, di *relazioni* tra i dati, e nessun calcolatore sembra ancora in grado di raggiungere vette così alte di complessità. Si pensi che il cervello umano contiene qualcosa come 10^{11} neuroni, e che ciascun neurone può essere connesso ad altri neuroni attraverso un migliaio di sinapsi, in modo che il numero totale di connessioni sinaptiche è dell'ordine di $10^{14} - 10^{15}$. Nei più sofisticati calcolatori, basati sull'integrazione a larga scala VLSI, il numero di elementi attivi (transistor) e di connessioni, che in una primissima e rozza approssimazione possiamo considerare equivalenti ai neuroni biologici e alle sinapsi, rimane confinato a un ordine di qualche unità per 10^9, e quindi c'è ancora un rapporto di almeno 10000:1 tra i due sistemi, quello biologico e quello artificiale. Ma naturalmente questo discorso vale solo in primissima approssimazione, poiché c'è anche da considerare che il cervello biologico non è un sistema rigidamente binario, e dunque per tener conto di questo fatto potrebbe essere necessario peggiorare ancora il rapporto visto prima.

L'informazione che riceviamo può in qualche caso essere in eccesso rispetto a quella strettamente necessaria per la comprensione del significato; in tal caso alla differenza fra le due si attribuisce il nome di *ridondanza*. Per esempio il completamento della frase "*Il primo mese dell'anno è Ge...*" non offre alcun tipo di difficoltà, poiché la conoscenza del significato in lingua italiana dei termini "primo", "mese" e "anno" porta univocamente a concludere che la parte mancante è "nnaio". Tuttavia, accanto a questa ricostruzione di tipo semantico, è possibile impostare anche una ricostruzione su base sintattica, basata sui semplici dati e senza nessun riferimento al loro significato. In tal caso la ridondanza si manifesta allorquando le lettere dell'alfabeto che si sta usando non hanno una probabilità uniforme di comparire. Si faccia riferimento alla tabella di figura 2.3, nella quale viene riportata la distribuzione delle frequenze relative delle lettere dell'alfabeto italiano nel romanzo *I promessi sposi*, considerati un riferimento per la nostra lingua. Si può notare che non tutte le lettere compaiono con la medesima frequenza, e anzi certe lettere sono piuttosto rare. Se approfondiamo l'analisi andando a calcolare le frequenze relative delle coppie, delle terne ecc. tipiche della lingua italiana, ci renderemo conto che la forbice fra combinazioni di lettere verosimili o anche solo possibili e quelle che non lo sono aumenta. Già l'analisi del secondo ordine ci mostra che alcune combinazioni sono vietate (p.es. la "qr" o la "mc" ecc), e altre sono esclusive, quali ad esempio la "qu", il che significa che una "q", in italiano, è quasi sempre seguita da una "u" (ci sono le eccezioni della doppia "q", come in "soqquadro" e "aqquartieramento").

E	0.12059	L	0.05567	V	0.02305
A	0.11512	S	0.05459	G	0.01713
O	0.09640	C	0.04692	H	0.01352
I	0.09530	D	0.03731	F	0.01052
N	0.07292	U	0.03569	B	0.00973
R	0.06599	P	0.02967	Q	0.00779
T	0.06083	M	0.02365	Z	0.00760

Figura 2.3: Tabella delle frequenze relative delle lettere in Italiano (ricavate da *I promessi sposi*).

Se si volesse ricostruire una frase basandosi su elementi sintattici, sarebbe allora necessario stimare la lettera (o il gruppo di lettere) che hanno la massima probabilità a priori di comparire subito dopo "Ge". Supponiamo che dalla lettura della tabella del terz'ordine emerga che la terna più probabile a priori che inizia con "Ge" sia "Ger". A questo punto dovremmo valutare la terna più probabile che inizia con "er", che supponiamo essere "ero", e così via. Il processo si può bloccare dopo aver raggiunto quella che può essere considerata una lunghezza tipica di una parola italiana. Se il risultato dell'esperimento di ricostruzione sintattica è la parola "Gerosa", è chiaro che la frase "*Il primo mese dell'anno è Gerosa*" è priva di significato in lingua italiana, ma tuttavia la parola ricostruita *sembra* essere una parola italiana, e ciò perché essa ha impressi i caratteri sintattici (statistici) della lingua italiana.

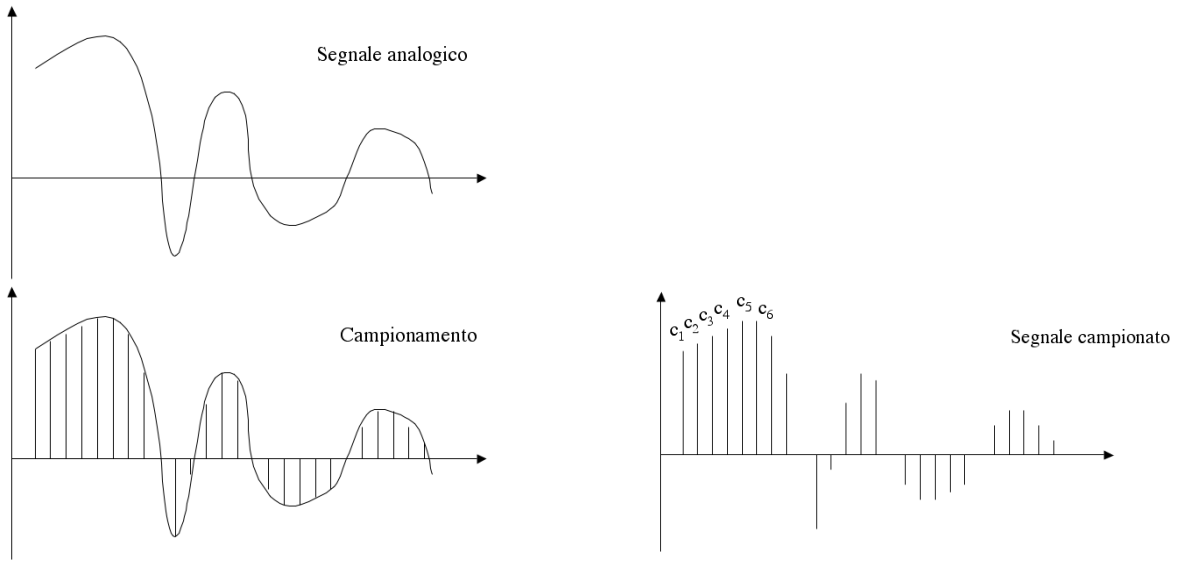
Riassumendo si può affermare che l'informazione non coincide col supporto, poiché può essere trasferita facilmente da un supporto all'altro, non segue le leggi di conservazione tipiche della fisica, poiché la sua distribuzione non la diminuisce, non è una grandezza fisica misurabile nel *Sistema Giorgi*, poiché è adimensionale, dipende dal contesto e dall'osservatore, almeno quando la si consideri per i suoi aspetti semantici. Si noti inoltre che anche l'assenza di informazione può essere informazione (0 e 1 sono tra loro alternatitivi e mutuamente escludentisi), poiché 0 è diverso da 1. L'informazione si sviluppa dunque nell'interazione tra supporto modulato dalle differenze e osservatore.

2.2 Informazione analogica e informazione discreta

L'informazione generata dalle varie sorgenti, impressa nei segnali che transitano sui vari canali, può avere tanto natura *continua* che *discreta*. Nel primo caso la grandezza fisica di riferimento cambia con continuità nel tempo, e il valore che assume istante per istante è dato da un numero reale. La precisione con la quale verrà poi letto questo valore in un certo istante, impiegando un opportuno strumento di misura, dipenderà essenzialmente dalla *classe* dello strumento, cioè dalle sue caratteristiche certificate di accuratezza e precisione.
Nel caso discreto, invece, l'informazione viene associata all'emissione di una lettera appartenente a un certo alfabeto finito, che potrebbe essere quello della lingua italiana nel caso di un testo di un romanzo, quello delle cifre da 0 a 9 nel caso si debbano fare dei calcoli, o quello della tastiera del computer (molto più ricco, e che comprende come sottoinsiemi i due alfabeti appena citati) nel caso si debba lavorare con dei programmi di videoscrittura.
La distinzione tra analogico e discreto (o *digitale*) si estende ovviamente anche ai sistemi elettronici che elaborano l'informazione, e si parla pertanto di sistemi *analogici* e di sistemi *digitali* (o *numerici*).

Nei primi decenni del novecento le apparecchiature per la manipolazione dell'informazione nelle sue varie forme (audio e video, principalmente) erano tutte analogiche (radio, televisori, telefoni ecc.); la parola *analogico* è stata coniata per ricordare che i segnali elettrici, che rappresentano le varie grandezze coinvolte nella riproduzione dell'informazione, seguono in modo *analogo* la grandezza fisica primitiva (la pressione acustica nel caso di un segnale audio, l'intensità luminosa nel caso di un segnale video ecc.). La manipolazione analogica dei segnali audio e video è coerente col fatto che questa informazione *nasce* in forma analogica, ed è del tutto ovvio gestirla usando lo stesso ambito.
L'informazione discreta era all'epoca confinata nel solo ambito specialistico e ristretto dei primi calcolatori, la cui logica di funzionamento, per quanto abbiamo anticipato nel paragrafo dedicato agli aspetti storici, era di tipo logico-matematico, e quindi basata su insiemi discreti di simboli.
Con l'andar del tempo si è però assistito a un progressivo trapasso dalle tecniche analogiche a quelle digitali, anche

in quegli ambiti dove l'informazione nasce in forma continua. Ciò è stato reso possibile dal celebre *teorema del campionamento* di Nyquist, risultato centrale nella teoria delle comunicazioni elettriche, che stabilisce la possibilità di *approssimare* un segnale analogico con uno discreto, con un errore asintoticamente nullo. Il campionamento prevede la lettura dei valori (campioni) del segnale analogico, nel suo sviluppo temporale, con una frequenza f_c denominata *frequenza di campionamento* (fig.2.4a). Il valore numerico espresso per ogni campione è già in forma



(a) Campionamento di un segnale analogico

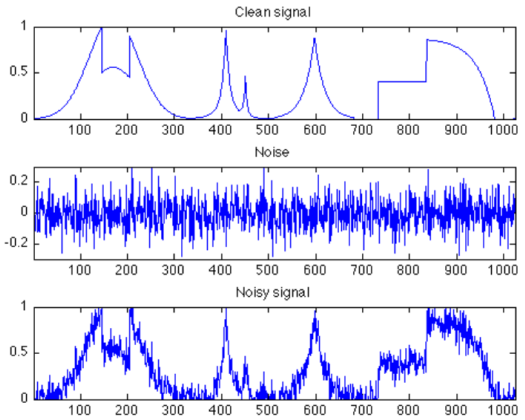
(b) Segnale campionato

Figura 2.4: Campionamento di un segnale analogico secondo il teorema di Nyquist

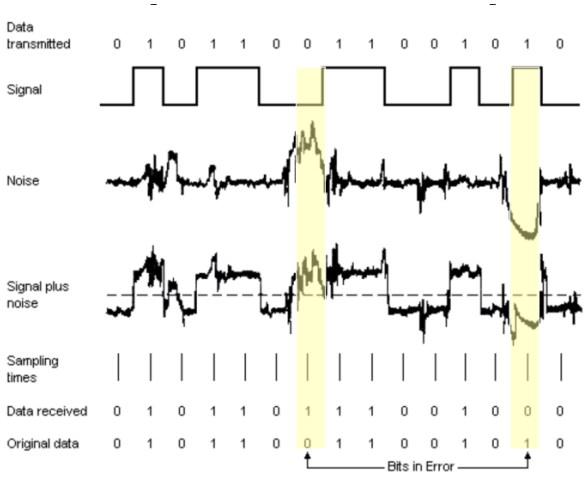
digitale, e se i campioni sono in numero sufficiente per ogni unità temporale (cioè se le frequenza di campionamento è sufficientemente alta), allora il teorema ci garantisce la possibilità di ricostruire in modo esatto il segnale continuo, senza altra perdita d'informazione che quella implicita nell'imprecisione della lettura (che porta al cosiddetto *rumore di campionamento*).
La possibilità di rielaborare i segnali convertiti in forma digitale mediante i processori, ha spinto a ricorrere sempre più frequentemente alle tecniche di *conversione analogico-digitale* (o conversione A/D); tale spinta è stata sostenuta anche dal fatto che, in genere, le apparecchiature digitali sono più affidabili di quelle analogiche, e ciò ha decretato di fatto l'abbandono della tecnologia analogica a favore di quella digitale in tutti i settori più importanti. Naturalmente nei casi in cui l'informazione nasca in forma analogica e debba poi essere riprodotta nella stessa veste, una volta effettuata la conversione A/D e le necessarie elaborazioni è necessario effettuare una successiva ri-*conversione digitale-analogica* (o conversione D/A), per poter recuperare il segnale analogico nella forma riproducibile presso l'utente, p.es. in quella di un segnale audio telefonico.

Un altro importante vantaggio delle apparecchiature digitali rispetto a a quelle analogiche è dato dalla loro resistenza al *rumore*. Ogni segnale elettrico gestito da circuiti elettronici viene irrimediabilmente inquinato da un segnale spurio denominato rumore, dato dalla sovrapposizione di numerosi segnali indotti, per la *legge di Lenz*, all'interno delle maglie chiuse dei circuiti elettronici. L'induzione avviene perché nello spazio nel quale siamo immersi esiste una moltitudine di onde elettromagnetiche dalle frequenze più disparate, sia di tipo artificiale (telefonia mobile, reti wi-fi, servizi radiofonici e televisivi, ...), sia di tipo naturale (radiazione solare, scariche atmosferiche, ...). La variazione del campo elettromagnetico esterno induce per la legge di Lenz una differenza di potenziale nel circuito interessato, che sviluppa una corrente spuria. Non dimentichiamo però che accanto al rumore di natura elettromagnetica c'è un'importante componente dovuta al *rumore termico* di agitazione degli elettroni, dovuto al fatto che i circuiti elettronici si trovano a una temperatura ambiente ben maggiore dello zero assoluto ($-273,16$ °C). Quando il segnale d'informazione viene colpito dal rumore, il segnale che ne deriva è dato dalla somma dei due, istante per istante. L'effetto è ben visibile in figura 2.5a e a questo punto non è più possibile separare i due segnali (a meno di casi particolari in cui stiano su bande diverse di frequenza); la compromissione

diviene irreversibile, e l'unico modo per preservare il contenuto informativo dall'inquinamento del rumore è quello di incrementare il rapporto segnale/rumore, amplificando il segnale d'informazione *prima* che venga attaccato dal rumore.



(a) Nei segnali analogici il rumore si somma al segnale in modo irreversibile

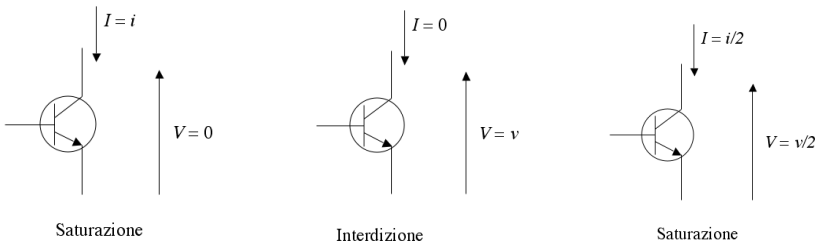


(b) Nei segnali digitali il segnale si può ricostruire in modo esatto, con una probabilità che può essere portata a valori prossimi a 1

Figura 2.5: Effetto del rumore sui segnali analogici e digitali

Viceversa, se un segnale associato a un alfabeto discreto viene compromesso dal rumore, è sempre possibile ricostruire l'unità elementare d'informazione (1 o 0) usando l'espediente di valutare l'area sottesa dal segnale intorno al suo valore medio (vedi figura 2.5b); il segnale discreto d'informazione potrà quindi essere ricostruito in modo esatto.
All'interno dei sistemi digitali si è poi assistito a uno sviluppo straordinario dei sistemi *binari*, motivati principalmente dall'esistenza dell'*algebra di Boole*, che come ricordato nel precedente capitolo ha fornito ai progettisti gli strumenti più idonei per progettare circuiti anche molto complessi. Si noti però che esistono anche motivazioni di carattere meramente tecnico che privilegiano la scelta di una base binaria rispetto altre basi ipotetiche, quali ad esempio quella ternaria, esadecimale o altro.
Come vedremo tra poco la scelta del binario comporta anche una maggiore affidabilità dei circuiti sui quali si basano le apparecchiature digitali.

A livello circuitale l'informazione binaria è associata allo stato di funzionamento dei dispositivi attivi (transistor), che sono in grado di controllare, tramite la base, il flusso di corrente che si sviluppa tra collettore ed emettitore. Contrariamente a quanto avviene per i *relè*, tale controllo è possibile con un dispendio minimo di energia e con continuità tra un valore minimo e un valore massimo. Un transistor correttamente polarizzato può trovarsi in uno stato di conduzione piena, o *saturazione* (fig.2.6a), in quello di *interdizione* (fig.2.6b) o in quello di conduzione parziale (fig.2.6c). Durante la saturazione del circuito collettore-emettitore la corrente I assume il suo valore massimo, mentre la tensione è nulla. Viceversa durante l'interdizione non scorre alcuna corrente, mentre la tensione V ai capi dei due elettrodi assume il suo valore massimo. Nello stato intermedio di conduzione parziale, entrambe le grandezze V e I sono diverse da zero. Se ora si vuole associare un'informazione binaria a un transistor, il modo più logico è quello di scegliere una delle grandezze che caratterizza il suo funzionamento, p.es. la tensione V, e considerare che il transistor sta nello stato logico 1 quando V assume il valore massimo, mentre sta nello stato logico 0 quando la V va a zero. Con questa convenzione lo stato di interdizione corrisponde allo stato 1, mentre quello di saturazione corrisponde allo stato 0. Se andiamo a valutare la potenza P_d dissipata dal transistor nelle due condizioni logiche 1 e 0 (interdizione e saturazione), ricordando che essa è data per la legge di *Joule* dal prodotto tra tensione e corrente, si avrà per la saturazione $P_d = V \cdot I = 0 \cdot I = 0$ e per l'interdizione $P_d = V \cdot I = V \cdot 0 = 0$. Ciò significa che in entrambi gli stati logici 0 e 1 *il transistor non dissipa potenza*. Se



Stato logico 0

$P_d = V\,I = 0\;i = 0$

(a) Nello stato logico 0 la tensione V e la potenza sono entrambe nulle

Stato logico 1

$P_d = V\,I = v\;0 = 0$

(b) Nello stato logico 1 la corrente I e la potenza sono entrambe nulle

Stato logico 2

$P_d = V\,I = v/2\;i/2 \neq 0$

(c) Nello stato logico 2 tensione, corrente e potenza sono diverse da zero

Figura 2.6: Potenza dissipata nei tre stati logici

ora volessimo introdurre un terzo stato logico, p.es. lo stato 2, per esso si dovrebbe individuare uno stato fisico di funzionamento del transistor che fosse massimamente distinguibile dagli stati logici 1 e 0, p.es. quello di una conduzione parziale di figura 2.6c, con tensione $V/2$ e corrente $I/2$. In questo caso, però, la potenza dissipata dal transistor sarebbe $P_d = V/2 \cdot I/2 \neq 0$, cioè maggiore di zero. Questo significherebbe la necessità di provvedere ad uno smaltimento del calore generato dal transistor, e dato che nei moderni circuiti integrati basati su una integrazione a larga scala ci sono, come ricordavamo nello scorso paragrafo, anche 10^9 transistor, si avrebbe un'insostenibile produzione di calore, con corrispondente aumento della temperatura di funzionamento delle giunzioni dei transistor, sulle quali si sviluppa la potenza. La temperatura è la principale nemica dei dispositivi a semiconduttore, poiché una sua deriva verso valori superiori a quelli massimi tollerati dal silicio, che sta intorno ai $180 - 200°$C, porta alla fusione della giunzione cristallina e alla distruzione del dispositivo. Ecco allora che queste motivazioni di carattere tecnologico, coniugate con lo sviluppo della logica Booleana, hanno portato al predominio assoluto dei dispositivi binari rispetto a qualunque altra soluzione. Anche i circuiti analogici, che vedono funzionare i transistor nelle condizioni di conduzione parziale, sono da questo punto di vista estremamente più critici e inaffidabili.
Concludendo questa analisi possiamo dire che il teorema del campionamento ci dà la possibilità di digitalizzare tutta l'informazione che nasce in forma analogica; la logica Booleana ci consente un'accurata progettazione dei circuiti binari; l'assenza di pericolose derive termiche legata alla scelta binaria aumenta in modo notevole la loro affidabilità. L'impiego sinergico di questo tre fattori ha di fatto decretato la sparizione delle tecnologie analogiche a favore di quelle digitali-binarie. Anche il nostro calcolatore, nato per altro per gestire informazione discreta, ha quindi una logica di funzionamento di tipo binario. Nei prossimi paragrafi dovremo allora occuparci di trasformare tutta l'informazione del mondo esterno in una modalità binaria.

2.3 L'alfabeto del calcolatore

Si è più volte sottolineata la natura simbolica degli oggetti che il calcolatore è in grado di gestire. I simboli in questione appartengono a un insieme, detto *alfabeto*, e ogni calcolatore può gestire uno o più alfabeti contemporaneamente. Può per esempio accettare in ingresso i simboli della tastiera immessi da un operatore, lavorare con un alfabeto binario $\mathbf{B} = \{0, 1\}$ a livello circuitale e magari rappresentare le informazioni che stanno sul disco rigido usando un alfabeto esadecimale del tipo $\mathbf{E} = \{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$. Tutti gli alfabeti visti finora hanno un numero finito di lettere, e si dicono *finiti*; per quanto esistano anche alfabeti infiniti, non avremo mai bisogno di usarli, e dunque tutte le considerazioni che faremo nel seguito riguarderanno solo alfabeti finiti.

L'esistenza delle diverse lettere è funzionale, più che altro, alla necessità di poter *distinguere* tra diversi

oggetti (o stati) che alle lettere vengono associati dal procedimento di codifica simbolica. Per fare un esempio banale potremmo codificare la posizione di una colonnina termometrica, che misuri le temperature di un ambiente condizionato, attribuendo all'intervallo $16 - 25°C$ una codifica nell'alfabeto $\mathbf{D} = \{0, 1, 2, \ldots, 8, 9\}$, in modo tale che 0 indichi 16, 1 indichi $17 \ldots 9$ indichi 25. In questo caso bisogna distinguere tra 10 stati, e possiamo usare le dieci cifre arabe. Il caso più semplice è quello in cui gli stati possibili sono solo due.
Per riprendere l'esempio della misura di una temperatura, potremmo riferirci alla situazione in cui sia sufficiente distinguere tra temperature sopra o sotto zero (lo zero lo poniamo per esempio in quest'ultima classe). In tal caso l'insieme che rappresenta i possibili stati consta di soli due elementi, che possiamo identificare convenzionalmente come $\mathbf{Z} = \{R, B\}$, dove R e B indicano rispettivamente i colori rosso e blu, tradizionalmente associati all'idea di caldo e freddo. Nessuno c'impedice però di usare altri simboli, per esempio $\mathbf{B} = \{0, 1\}$, poichè l'importante non è tanto il simbolo di per sé, che ha natura convenzionale, quanto piuttosto il numero di stati possibili da rappresentare, che in questo caso sono due. Si noti che la distinzione tra due stati diversi è minimale, e dunque l'alfabeto binario è il più piccolo alfabeto che si possa impiegare per distinguere tra situazioni diverse. Gli alfabeti che si usano poi effettivamente dipendono in modo essenziale dal contesto d'uso. Per esempio è ovvio usare l'alfabeto italiano di 21 simboli per scrivere frasi in tale lingua, mentre se si passa alla lingua inglese sarà necessario usare un alfabeto più ricco, contenente 26 simboli (in inglese ci sono le lettere j, k, x, y, w che mancano in italiano).

E' interessante osservare che, per quanto gli alfabeti possano essere apparentemente molto diversi l'uno dall'altro, la loro essenza simbolica ci consente di trasformare un alfabeto in un altro, ricorrendo sempre alla tecnica della codifica. In tal modo un alfabeto quaternario del tipo $Q = \{A, B, C, D\}$ può essere espresso nei termini di un alfabeto binario, per esempio usando coppie di cifre binarie, ottenendo $Q = \{00, 01, 10, 11\}$. Si noti che in tal caso si è di fatto riusciti a esprimere un alfabeto quaternario usandone uno binario. Poiché le due cifre $0, 1$ non bastano per distinguere tra i quattro stati A, B, C, D, si è ricorso al semplice espediente di impiegare più cifre binarie, facendo aumentare le combinazioni con le quali quest'ultime possono essere organizzate. Anche quest'aspetto verrà approfondito nel seguito; per ora ci basti riflettere sulla circostanza, solo apparentemente banale, che *con un alfabeto binario si può rappresentare potenzialmente un qualunque alfabeto (finito), pur di usare blocchi binari di lunghezza adeguata*.
Nella precedente sezione 2.2 abbiamo giustificato ampiamente il fatto che i calcolatori sono costretti a lavorare con un alfabeto binario a livello circuitale. Se dunque le informazioni del mondo esterno di figura 2.1 devono essere codificate in stringhe binarie, bisogna trovare una modalità per effettuare la codifica qualunque sia la tipologia di informazione con la quale abbiamo a che fare, analogica o discreta che sia.

Sulla base della citata figura possiamo distinguere tra quattro tipologie diverse d'informazione che possiamo accettare sui dispositivi o *periferiche* d'ingresso del computer:

- *Lettere e simboli grafici* vari derivanti dalla tastiera, dalla connessione *Internet*, da memorie esterne quali chiavette USB, ecc. (informazione discreta)
- *Numeri* usati per effettuare calcoli (informazione discreta) quando si impiega software matematico
- *Segnali analogici* di varia natura che costituiscono il segnale d'uscita di un *trasduttore* che trasforma una grandezza fisica che vogliamo rilevare (temperatura, pressione acustica, luminosità, contrasto, ...) in un segnale elettrico (tensione, corrente, resistenza, capacità, induttanza, ...)
- *Immagini* acquisite con una telecamera (informazione analogica)

2.3.1 Lettere e simboli grafici dalla tastiera

Consideriamo ora il primo caso, quello dei simboli della tastiera. È noto che bisogna distinguere tra lettere maiuscole e minuscole (A, a, B, b, ...), segni di interpunzione (, . : ...), numeri (0, 1, 2, ..., 9), ma ci sono anche dei simboli speciali, quale il tasto "invio", il tasto "tabulatore", ecc. Si noti tuttavia che questi simboli potrebbero non derivare direttamente dall'operatore che sta alla tastiera, poiché potrebbero essere i simboli di un generico file di testo che è memorizzato nel nostro disco rigido e che vogliamo rielaborare, oppure che è memorizzato su un

supporto esterno di memoria, o ancora che deriva dalla ricezione di un pacchetto dati dalla rete *Internet* mediante un cavo o mediante *wi-fi*. Per mantenere il nostro discorso a un livello astratto, chiamiamo *sorgente d'informazione* il dispositivo (disco rigido, chiavetta USB, porta Ethernet...) che genera la sequenza di simboli nel nostro alfabeto di riferimento. Chiamiamo allora *codifica di sorgente* la traduzione dall'alfabeto della sorgente, che in generale può essere assimilato all'alfabeto di un linguaggio naturale, all'alfabeto di funzionamento del sistema digitale (calcolatore, canale di trasmissione, memoria,...), che è legato a considerazione di carattere tecnico-operativo, per esempio alla circostanza che la tecnologia elettronica, che incarna il funzionamento dei varî dispositivi, è essenzialmente legata a una logica binaria, e quindi a un alfabeto di due simboli.
Sia allora $\mathcal{A} = \{a_1, a_2, \ldots, a_K\}$ l'alfabeto della sorgente, o alfabeto *primario*, e $\mathcal{B} = \{b_1, b_2, \ldots, b_D\}$ l'alfabeto sul quale si basa il funzionamento del sistema digitale, o alfabeto *secondario*. Di solito $D = 2$ e $K > D$. La traduzione tra le sequenze sui due alfabeti avviene a opera di un dispositivo chiamato *codificatore*; essa viene attuata in modo tale che dalla sequenza secondaria che esce dal *codificatore* si riesca a ricavare in modo univoco la sequenza primaria generata dalla sorgente; tale condizione si esprime affermando che il codice dev'essere *univocamente decodificabile* (u.d.). È ovvio che l'ipotesi di univoca decodificabilità risulta irrinunciabile.
Esigenze di carattere economico richiedono, in qualche caso, che la traduzione sia accompagnata da una efficienza che si può misurare sulla base del rapporto tra lunghezza (media) della sequenza secondaria e lunghezza (media) della sequenza primaria. Questa efficienza non è di per sé irrinunciabile, ma sicuramente fortemente auspicabile. Infatti l'occupazione della memoria o del canale di trasmissione ha un costo che si può assimilare proporzionale alla lunghezza della sequenza secondaria. Il problema generale da risolvere, nell'ambito della codifica di sorgente, è allora quello di individuare una strategia di traduzione u.d. che comporti un'elevata efficienza, cioè un'elevata *compressione dei dati*. In qualche caso, però, è sufficiente effettuare una semplice *traduzione* tra i due alfabeti, senza cioè richiedere una compressione dati. E' questo il caso dei simboli della tastiera, quando vengono immessi codificati sotto forma binaria all'interno del calcolatore.
Passiamo ora alla descrizione del modello vero e proprio [9]. Sia $\mathcal{A}^+$ l'insieme di tutte le sequenze finite (stringhe) costruite con elementi di $\mathcal{A}$ e $*$ la legge di composizione interna per i suoi elementi data dalla *concatenazione* tra gli stessi; così se $\boldsymbol{\alpha_1}, \boldsymbol{\alpha_2} \in \mathcal{A}^+$, anche $\boldsymbol{\alpha_1} * \boldsymbol{\alpha_2} \in \mathcal{A}^+$. Se consideriamo ora l'insieme $\mathcal{B}^+$ associato all'alfabeto $\mathcal{B}$ delle sequenze secondarie si definisce *codice astratto* una qualunque applicazione

$$\varphi : \mathcal{A}^+ \to \mathcal{B}^+$$

Questa definizione è però troppo generale e porta a un codice impraticabile, poiché pone in corrispondenza gli elementi di due insiemi infiniti. Tuttavia essa può essere circostanziata al caso in cui la φ sia un *omomorfismo*; se ciò accade si ha una semplificazione notevole, poiché in tal caso $\varphi(\boldsymbol{\alpha_1} * \boldsymbol{\alpha_2}) = \varphi(\boldsymbol{\alpha_1}) * \varphi(\boldsymbol{\alpha_2})$ (si assume che anche $\mathcal{B}^+$ abbia come legge di composizione interna la $*$). In altre parole la *codifica* di una sequenza di stringhe primarie concatenate si ottiene mediante *concatenazione delle codifiche* (basate sull'alfabeto secondario) delle singole stringhe primarie. Come conseguenza di ciò verificheremo che è sufficiente stabilire una corrispondenza tra gli elementi di due insiemi finiti, quello dei *messaggi*, $\mathcal{M} = \{m_1, m_2, \ldots, m_{_T}\}, m_i \in \mathcal{A}^+$, che contiene stringhe (eventualmente a lunghezza variabile) costruite sulle lettere di $\mathcal{A}$, e l'insieme dei *valori* (o *parole di codice*) che la φ fa assumere a tali messaggi. Ecco allora che accanto all'insieme primario $\mathcal{M} = \{m_i\}_{i=1}^{K}$ considereremo il *dizionario*

$$\mathcal{W} = \{\varphi(m_i)\}_{i=1}^{K} \quad \varphi(m_i) \in \mathcal{B}^+$$

delle *parole di codice* (eventualmente a lunghezza variabile) associate a $\mathcal{M}$. In particolare i messaggi possono ridursi a singole lettere, e in questo caso il dizionario rappresenta la codifica dell'alfabeto primario; la costruzione di $\mathcal{M}$ è allora immediata, poiché $\mathcal{M} = \mathcal{A}$.

Definizione 2.1. *Si definisce codice la terna* $\mathcal{C} = \{\mathcal{M}, \varphi, \mathcal{W}\}$.

Si noti che la costruzione di $\mathcal{M}$ non è in generale banale: bisogna infatti fare in modo che con i suoi elementi si possa comporre una qualunque stringa $\boldsymbol{\alpha} \in \mathcal{A}^+$; ciò equivale a dire che qualunque sia la sequenza $\boldsymbol{\alpha}$ emessa dalla sorgente deve essere possibile effettuare una sua *segmentazione* negli elementi di $\mathcal{M}$.

Definizione 2.2. *Una famiglia* $\mathcal{M} = \{m_1, m_2, \ldots, m_{_T}\}$ *di messaggi si dice esauriente se, qualunque sia la successione* $\boldsymbol{\alpha}$ *semi-infinita a destra costruita con gli elementi di* $\mathcal{A}$*, esiste sempre almeno un prefisso di* $\boldsymbol{\alpha}$ *che appartiene a* $\mathcal{M}$.

L'univoca decodificabilità può essere introdotta imponendo *l'iniettività* della funzione di codifica φ

Definizione 2.3. *Un codice* $\mathcal{C} = \{\mathcal{M}, \varphi, \mathcal{W}\}$ *è univocamente decodificabile se,* $\forall\, \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathcal{A}^+$, $\boldsymbol{\alpha} \neq \boldsymbol{\beta}$ *implica* $\varphi(\boldsymbol{\alpha}) \neq \varphi(\boldsymbol{\beta})$.

Per quanto attiene i codici di sorgente distingueremo gli stessi sulla base della lunghezza (costante o variabile) dei messaggi e delle parole di codice. Ci sono dunque quattro possibilità:

Codici blocco-blocco (*B-B*). In questo caso i messaggi si riducono a blocchi di lunghezza costante pari a n, cioè $\mathcal{M} = \mathcal{A}^n$ per $n \geqslant 1$; anche la lunghezza delle parole di codice è costante;

Codici blocco-lunghezza variabile (*B-LV*). Le parole di codice $\varphi(m_i)$ sono a lunghezza variabile, mentre i messaggi hanno lunghezza costante pari a n. Molto spesso $n = 1$, e la codifica a lunghezza variabile riguarda le singole lettere di $\mathcal{A}$. Se $n > 1$ allora i messaggi sono tutte le possibili K^n n-ple di $\mathcal{A}^n$.

Codici lunghezza variabile-blocco (*LV-B*). I messaggi sono a lunghezza variabile e bisogna garantire l'esaurienza. Le parole di codice sono invece a lunghezza costante.

Codici lunghezza variabile-lunghezza variabile (*LV-LV*). È il caso più generale: a ciascun messaggio primario di lunghezza variabile si fa corrispondere una parola di codice anch'essa di lunghezza variabile.

Esempio 2.1. Sia $\mathcal{A} = \{a, b, c\}$ e $\mathcal{B} = \{0, 1\}$. Effettuiamo una codifica per ciascuno dei quattro casi precedenti, supponendo che la sequenza emessa sia $\boldsymbol{\alpha} = bbacab$.

B-B Sia $\mathcal{W} = \{\varphi(a), \varphi(b), \varphi(c)\} = \{00, 01, 10\}$. La codifica va fatta sulle singole lettere: $\varphi(bbacab) = 01/01/00/10/00/01$.

B-LV Se $\mathcal{W} = \{\varphi(a), \varphi(b), \varphi(c)\} = \{0, 10, 11\}$, allora $\varphi(bbacab) = 10/10/0/11/0/10$.

LV-B In questo caso bisogna costruire una famiglia di messaggi a lunghezza variabile; poniamo p.es. $\mathcal{M} = \{aa, ab, ac, b, c\}$. La funzione di codifica associa ai messaggi parole di codice a lunghezza costante, p.es. $\mathcal{W} = \{\varphi(aa), \varphi(ab), \varphi(ac), \varphi(b), \varphi(c)\} = \{000, 001, 010, 011, 100\}$. La segmentazione della sequenza primaria è allora $b/b/ac/ab$ e la successione delle parole di codice emesse dal codificatore è $011/011/010/001$.

LV-LV Manteniamo la stessa famiglia di messaggi vista prima $\mathcal{M} = \{aa, ab, ac, b, c\}$. La funzione di codifica associa a ciascun messaggio parole di codice a lunghezza variabile, p.es. $\mathcal{W} = \{\varphi(aa), \varphi(ab), \varphi(ac), \varphi(b), \varphi(c)\} = \{000, 001, 01, 10, 11\}$. Con la stessa segmentazione in messaggi del caso precedente si ottiene la seguente successione secondaria: è $10/10/01/001$. ◯

Dall'esempio si noti come, nel caso in cui le parole di codice siano a lunghezza costante, l'omomorfismo della funzione di codifica sposti la richiesta di univoca decodificabilità dalle sequenze alle singole parole di codice; in tal caso è infatti sufficiente fare in modo che esse siano tutte distinte.

Osservazione 2.1. Un discorso analogo vale anche per la *segmentazione* delle sequenze primarie nel caso di codici con messaggi a lunghezza variabile; la principale differenza operativa consiste nel fatto che in questo caso una segmentazione non univoca (mancanza di univoca *codificabilità*) non è pregiudizievole, poiché si può scegliere una delle due (o più) segmentazioni secondo opportuni criteri di convenienza.

Osservazione 2.2. Mentre la condizione u.d. è irrinunciabile per il dizionario di una codifica *B-LV*, la sua esaurienza non è rilevante, poiché la decodifica va fatta su sequenze costruite con elementi dello stesso dizionario. Viceversa l'esaurienza è strettamente necessaria per la famiglia di messaggi in una codifica *LV-B*, poiché dalla sorgente può uscire una sequenza qualunque. Per contro, come riferito nell'osservazione precedente, si può operare anche in assenza di univoca codificabilità.

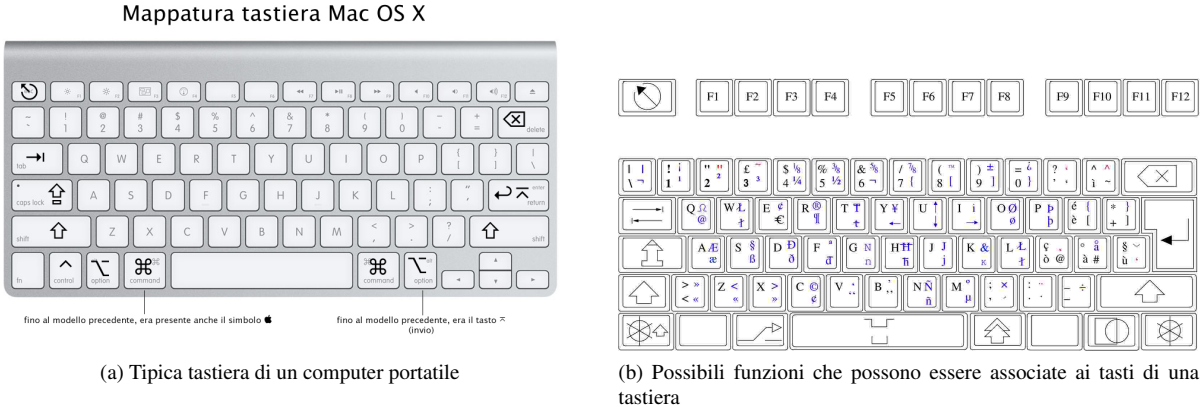
Nel caso dell'alfabeto di una tastiera, la prima necessità è quella di eseguire una semplice traduzione in binario, in modo che le lettere siano disponibili in forma binaria ai circuiti interni del calcolatore; in questo caso siamo chiaramente nell'ambito dii una codifica B-B per lettere singole. Si potrebbe dimostrare che la codifica B-B non consente di effettuare una compressione dati, e si tratta dunque di una semplice traduzione tra due alfabeti. Per comprimere dati bisogna ricorrere a una codifica a lunghezza variabile, sulla quale si basano le varie procedure usate per questo scopo (*zip, pkzip, compress, compact*, ...); non tratteremo però questa parte, perché esula dagli obiettivi del corso.
Supponiamo dunque che K sia il numero dei simboli della tastiera e che si effettui una codifica a *lunghezza costante*, in modo che ciascuna stringa abbia una lunghezza n. Il problema è allora quello di calcolare il valore di n. Quante sono le possibili n-ple, cioè le stringhe binarie di lunghezza n?
Per capirlo prendiamo $n = 1$. Se abbiamo una sola cifra binaria essa può essere 0 o 1, e dunque 2 possibilità. Se prendiamo due cifre binarie consecutive ci sono 4 possibilità, 00, 01, 10, 11; ciò accade perché per ciascuna delle 2 possibilità per la prima cifra ci sono 2 possibilità per la seconda, e quindi $2 \cdot 2 = 2^2 = 4$. Per $n = 3$ abbiamo, con un ragionamento analogo, $2 \cdot 2 \cdot 2 = 2^3 = 8$. Nel caso generico n ci sono allora $2 \cdot 2 \cdot \ldots \cdot 2$ moltiplicato per n volte, cioè 2^n. Se vogliamo che sia garantita l'ipotesi di univoca decodificabilità dobbiamo avere almeno una n-pla per ciascun simbolo della tastiera, e dunque deve valere la relazione

$$2^n \geqslant K$$

Applicando il logaritmo in base 2 e risolvendo rispetto a n, che deve essere un numero intero, si ottiene

$$n \geqslant \log_2 K \qquad \text{cioè} \qquad n = \lceil \log_2 K \rceil$$

dove il simbolo $\lceil x \rceil$ significa la *parte intera superiore* di x, cioè il più piccolo intero maggiore o uguale a x. Per esempio $\lceil 3,2 \rceil = 4$, $\lceil 5,99 \rceil = 6$, $\lceil 6 \rceil = 6$, $\lceil 3,001 \rceil = 4$. Analogamente per il simbolo $\lfloor x \rfloor$, che rappresenta la *parte intera inferiore* di x, cioè il più grande intero minore o uguale a x.
Se ora prendiamo la tastiera di figura 2.7a e contiamo quanti sono i simboli di base troviamo $26 \cdot 2$ lettere, $10 \cdot 2$ simboli sui tasti numerici, 22 segni di interpunzione e altri di vario genere e 17 altri tasti speciali, per un totale di 111 simboli. Poiché $\lceil \log_2 111 \rceil = 7$, con 7 bit si possono costruire $2^7 = 128 > 111$ possibili n-ple binarie



(a) Tipica tastiera di un computer portatile

(b) Possibili funzioni che possono essere associate ai tasti di una tastiera

Figura 2.7: Tastiere e funzioni di una tastiera

e nel 1961 un ingegnere della *IBM* propose una tabella di codifica, rappresentata in figura 2.8, che divenne successivamente lo standard ASCII (*American Standard Code for Information Interchange*). Sette bit non sono però sufficienti per le lingue più comuni, poiché ci sono anche le lettere accentate, le dieresi, le cediglie ecc. La figura 2.7b ci illustra per esempio tutte le funzioni che si possono ricavare dalla tastiera di un moderno computer portatile. Si può notare che per ogni tasto ci sono quasi sempre 3 o 4 funzioni diverse, e questo porta ben oltre 128 il numero delle possibili combinazioni. Tant'è che subito dopo l'introduzione dell'ASCII, venne proposta un'estensione di un bit, per portare il codice a $8\,\text{bit} = 1\,\text{byte}$. Ciò consente di raddoppiare i simboli, passando da 128 a 256 possibili combinazioni e di inserire anche le lettere con accenti speciali delle varie lingue europee. Si pervenne così a un

Bits b_4 b_3 b_2 b_1	Row ↓ / Column → 0	1	2	3	4	5	6	7
	b_7 b_6 b_5 0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FC	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Figura 2.8: Tabella del codice ASCII. La 7-pla binaria che corrisponde a ciascun simbolo si legge come $b_7b_6b_5b_4b_3b_2b_1$. P.es. la codifica di 9 è 0111001, mentre quella di m è 1101101

insieme di standard, denominati ISO 8859-n, nei quali le prime 128 codifiche coincidono con quelle dell'ASCII standard, mentre l'estensione degli ulteriori 128 bit dipende dal valore di n (compreso tra 1 e 16). Per esempio ISO 8859-1 contiene le lettere speciali delle principali lingue europee nord-occidentali ($\tilde{n}$, ü, å, è, é, ß ecc.), mentre ISO 8859-5 contiene simboli del cirillico, ISO 8859-8 l'ebraico e così via (fig.2.9). Tuttavia questi standard non erano seguiti da tutti i costruttori, per cui poteva accadere che passando da una piattaforma *hardware* a un'altra ci fossero dei problemi con la decodifica. Attualmente la codifica è basata sullo standard perfezionato dall'*Unicode Consortium*, un consorzio di aziende interessate all'unificazione delle codifiche a livello internazionale, che possa comprendere tutte le lingue parlate nel mondo. Partito nella versione a 16 bit, che consente 65536 possibili combinazioni, viene ora proposto nelle versioni a 32 bit con sottocodifiche denominate *Unicode Transformation Format* del tipo UTF-8, UTF-16 e UTF-32. Le prime due codifiche sono a lunghezza variabile, nel senso che UTF-8 usa da 1 a 4 byte per rappresentare i vari simboli, mentre UTF-16 ne usa 2 o 4. Esse sono più efficienti, in termini di spazio occupato, della versione UTF-32, che necessita di 32 bit per tutti i simboli.
Con questo tipo di codifica, le cui tabelle di corrispondenza non sono ancora state completate, si riescono a rappresentare i caratteri di tutte le lingue vive e di molte lingue morte, ma anche i caratteri di scritti del patrimonio storico dell'umanità, nelle diverse lingue e negli svariati sistemi di segni utilizzati nel passato.
La codifica ASCII appena analizzata e le sue generalizzazioni più recenti, quali l'Unicode, consentono di codificare in binario qualunque simbolo venga immesso dalla tastiera o provenga dalla rete *Internet* attraverso il cavo di connessione, oppure attraverso un collegamento *wi-fi*.

2.3.2 La rappresentazione dei numeri

Quando scriviamo un indirizzo su una lettera, p.es. *via Roma 51*, usiamo le cifre 5 e 1 come semplici simboli grafici. Se vogliamo invece eseguire la somma $51 + 34 = 85$, l'uso dei simboli numerici 5 e 1 sottintende un impiego diverso delle cifre, legato alle regole dell'aritmetica dei *numeri naturali*, indicati in matematica col simbolo $\mathbb{N}$; essi sono $0, 1, 2, 3, 4, ...$, cioè infiniti. Anche se non possiamo sperare di rappresentare *tutti* i numeri naturali, vista la loro infinitezza, sarebbe comunque logico pensare a una codifica diversa per un numero naturale

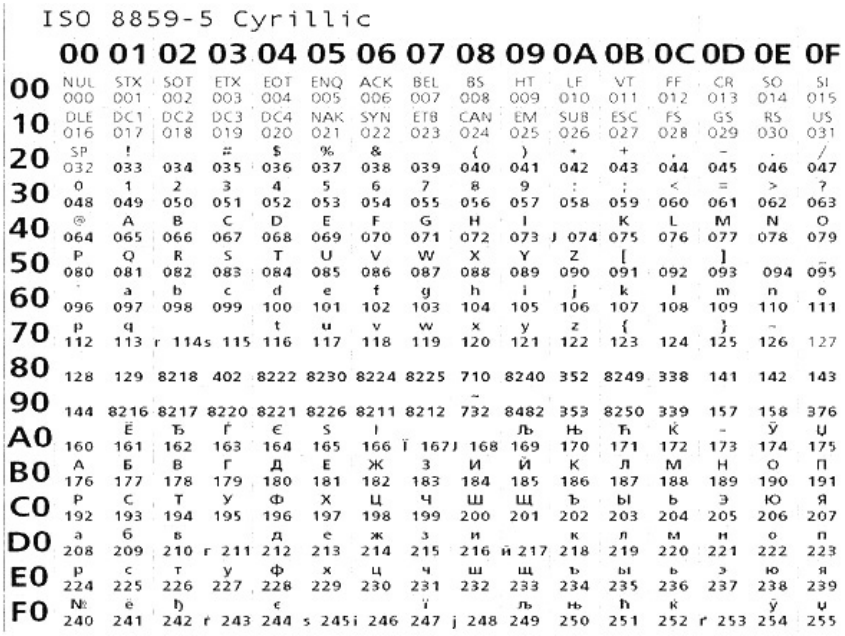


Figura 2.9: Estensione ISO 8859-5 del codice ASCII, che codifica le lettere dell'alfabeto cirillico

quando questo debba essere impiegato per i calcoli. Poiché all'interno di un computer si lavora in binario, riprendendo l'esempio della somma di prima l'ideale sarebbe che la codifica (binaria) del 51 *sommata* alla codifica del 34 fornisse la codifica del numero 85. Se vogliamo tentare questo approccio dobbiamo fare qualche passo indietro e pensare alla notazione che usiamo fin da piccoli per i numeri decimali, che ha il pregio di consentirci di fare agevolmente le quattro operazioni.

La rappresentazione dei numeri interi: la notazione posizionale

Quando scriviamo il numero 357 lo leggiamo come *trecentocinquantasette*; è uno schema mentale automatico, che ci sembra normale e che abbiamo acquisito nella scuola elementare. Si tratta della cosiddetta *notazione posizionale*, introdotta in Cina due secoli prima di Cristo e perfezionata in India nel 500. In Europa verrà usata solo a partire dal Rinascimento, e infatti la numerazione romana, di cui conserviamo ancora qualche traccia, è una notazione non-posizionale con la quale è estremamente complesso far di conto.
Notazione posizionale significa che il *valore* di ciascuna cifra $0..9$ dipende dalla *posizione* nella quale si trova all'interno del numero. Di conseguenza 357 significa 3 centinaia, 5 decine e 7 unità, che scritto in modalità matematica usando le potenze di 10 porta a

$$357 = \mathbf{3} \cdot 10^2 + \mathbf{5} \cdot 10^1 + \mathbf{7} \cdot 10^0$$

La notazione sottintende che stiamo usando la base 10, ma naturalmente si potrebbe usare anche una qualunque altra base b per rappresentare un numero intero come

$$\boldsymbol{a_n a_{n-1} ... a_0} = \boxed{\boldsymbol{a_n}} \cdot b^n + \boxed{\boldsymbol{a_{n-1}}} \cdot b^{n-1} + ... + \boxed{\boldsymbol{a_0}} \cdot b^0 \tag{2.1}$$

con la convenzione che ogni a_i deve essere compreso tra 0 e $b-1$ (tra 0 e 9 nella base $b = 10$). Il vantaggio della notazione posizionale è quello relativo alle regole elementari per effettuare le quattro operazioni, quali la somma e la moltiplicazione in colonna.

La conversione da base 2 a base 10 e viceversa Se vogliamo usare un alfabeto binario per rappresentare i numeri, possiamo pensare di usare una base $b = 2$ nella rappresentazione posizionale 2.1. In questo modo ciascuna cifra a_i sarà compresa tra 0 e 1, e dunque binaria. Con questa convenzione il numero binario 101001 corrisponde a

$$\mathbf{101001} = \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{1} \cdot 2^0$$

Per capire di che numero decimale si tratta è sufficiente sviluppare i calcoli degli esponenti e sommare i termini, ottenendo $32 + 8 + 1 = 41$, che corrisponde a una *conversione* da base 2 a base 10.
Se vogliamo invece fare la conversione opposta, da base 10 a base 2, bisogna rielaborare la formula 2.1, espressa in base 2, che per comodità riportiamo sotto

$$\boldsymbol{a_n a_{n-1} \ldots a_0} = \boldsymbol{a_n} \cdot 2^n + \boldsymbol{a_{n-1}} \cdot 2^{n-1} + \ldots + \boldsymbol{a_2} \cdot 2^2 + \boldsymbol{a_1} \cdot 2^1 + \boldsymbol{a_0} \cdot 2^0 \tag{2.2}$$

In questo caso partiamo dal numero decimale, per esempio $N = 357$, e dobbiamo individuare gli elementi binari $\boldsymbol{a_n a_{n-1} \ldots a_0}$ in modo che espressi nella notazione 2.2 rappresentino 357 in decimale. In pratica i coefficienti $\boldsymbol{a_n a_{n-1} \ldots a_0}$ sono le nostre incognite. Per individuarle possiamo fare in questo modo; mettiamo in evidenza un 2 dal blocco della notazione

$$\begin{aligned} N &= (\boldsymbol{a_n} \cdot 2^{n-1} + \boldsymbol{a_{n-1}} \cdot 2^{n-2} + \ldots + \boldsymbol{a_2} \cdot 2^1 + \boldsymbol{a_1}) \cdot 2 + \boldsymbol{a_0} \\ &= \boldsymbol{q_1} \cdot 2 + \boxed{\boldsymbol{a_0}} \end{aligned}$$

Si nota che l'incognita $\boldsymbol{a_0}$ risulta essere il resto della divisione di N per 2. Se continuiamo in questo modo possiamo poi scrivere $\boldsymbol{a_1}$ come resto della divisione di $\boldsymbol{q_1}$ per 2, $\boldsymbol{a_2}$ come resto della divisione di $\boldsymbol{q_2}$ per 2 e così via:

$$\begin{aligned} \boldsymbol{q_1} &= (\boldsymbol{a_n} \cdot 2^{n-2} + \boldsymbol{a_{n-1}} \cdot 2^{n-3} + \ldots + \boldsymbol{a_2}) \cdot 2 + \boldsymbol{a_1} \\ &= \boldsymbol{q_2} \cdot 2 + \boxed{\boldsymbol{a_1}} \\ \\ \boldsymbol{q_2} &= (\boldsymbol{a_n} \cdot 2^{n-3} + \boldsymbol{a_{n-1}} \cdot 2^{n-4} + \ldots + \boldsymbol{a_3}) \cdot 2 + \boldsymbol{a_2} \\ &= \boldsymbol{q_3} \cdot 2 + \boxed{\boldsymbol{a_2}} \\ &\vdots \\ \boldsymbol{q_n} &= 0 \cdot 2 + \boxed{\boldsymbol{a_n}} \end{aligned}$$

Questo procedimento risolve il nostro problema della conversione da base 10 a base 2. Proviamo col 537

$$\begin{aligned} 537 &= 268 \cdot 2 + \boxed{1} && \Uparrow \\ 268 &= 134 \cdot 2 + \boxed{0} \\ 134 &= 67 \cdot 2 + \boxed{0} \\ 67 &= 33 \cdot 2 + \boxed{1} && \Uparrow \\ 33 &= 16 \cdot 2 + \boxed{1} \\ 16 &= 8 \cdot 2 + \boxed{0} \\ 8 &= 4 \cdot 2 + \boxed{0} && \text{lettura dal bit più significativo} \\ 4 &= 2 \cdot 2 + \boxed{0} && \text{a quello meno significativo} \\ 2 &= 1 \cdot 2 + \boxed{0} \\ 1 &= 0 \cdot 2 + \boxed{1} && \Uparrow \end{aligned}$$

Poiché il primo bit che si ottiene dal procedimento è $\boldsymbol{a_0}$, cioè quello meno significativo, la lettura della stringa binaria deve essere fatta dal basso verso l'alto, il che porta al numero 1000011001.
Se ora vogliamo verificare che la conversione sia stata fatta correttamente, è sufficiente fare la riconversione da base 2 a base 10

$$\begin{aligned} \mathbf{1000011001} &= \mathbf{1} \cdot 2^9 + \mathbf{1} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{1} \\ &= 512 + 16 + 8 + 1 = 537 \end{aligned}$$

La notazione posizionale in base 2 con n bit ci consente di rappresentare tutti i 2^n numeri interi (positivi) compresi tra 0 e $2^n - 1$. Se disponiamo invece di una generica base b, l'intervallo di rappresentazione va da 0 a $b^n - 1$.
Se prendiamo per esempio $n = 4$ nel caso binario, otteniamo la rappresentazione dei numeri da 0 a $2^4 - 1 = 15$ evidenziata in figura 2.10a. Grazie all'impiego della notazione posizionale possiamo ora fare somme e moltiplica-

0111	7	1111	15
0110	6	1110	14
0101	5	1101	13
0100	4	1100	12
0011	3	1011	11
0010	2	1010	10
0001	1	1001	9
0000	0	1000	8

(a) Rappresentazione posizionale in base 2 dei numeri da 0 a $2^4 - 1$

Somma	Prodotto
$0 + 0 = 0$	$0 \cdot 0 = 0$
$0 + 1 = 1$	$0 \cdot 1 = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$
$1 + 1 = 0$	$1 \cdot 1 = 1$
con riporto	

(b) Somma e prodotto in base 2

Figura 2.10: Rappresentazione binaria e operazioni relative

zioni tra numeri binari usando le stesse regole usate per i numeri decimali. La figura 2.10b ci mostra le operazioni di base; si noti che $1+1 = 10$ e dunque, in una somma in colonna, bisogna effettuare il riporto di 1 esattamente come in decimale, quando $1 + 9 = 10$, scrivo 0 e riporto 1. Nella figura 2.11a sono invece riportate una somma e una moltiplicazione eseguite in colonna, secondo le regole tradizionali. Con l'espediente della notazione posizionale si realizza l'obiettivo iniziale di una codifica della somma (del prodotto) che corrisponde alla somma (prodotto) delle codifiche. Quando si deve gestire l'informazione che deriva da una codifica in binario, p.es. quella associata

$$\begin{array}{cccc} & {\scriptstyle 1} & & \\ 1010 & + & 10 & \\ \underline{0011} & = & \underline{3} & \\ 1101 & & 13 & \end{array} \qquad \begin{array}{cccc} & & & \\ 0100 & * & 4 & \\ \underline{0011} & = & \underline{3} & \\ 0100 & & & \\ \underline{0100} & & \underline{\quad} & \\ 01100 & & 12 & \end{array}$$

(a) Somme e prodotti in base 2 si eseguono secondo le regole consuete

$$\begin{array}{cccc} & {\scriptstyle 1} & & \\ & 1 & 0 & 1 & 0 & 10 & + \\ & 1 & 1 & 0 & 0 & \underline{12} & = \\ 1 & 0 & 1 & 1 & 0 & 6 & \end{array}$$

(b) Somma che porta a un errore di *overflow*

Figura 2.11: Esempi di operazioni in base 2

a un simbolo ASCII o alla codifica in notazione posizionale di un numero, bisogna assegnare a priori una certa risorsa di memoria all'oggetto che stiamo manipolando. Per esempio se usiamo un codice ASCII esteso ISO 8859 assegneremo sempre 8 bit, cioè 1 byte a ciascun simbolo, mentre se usiamo un UTF-32 assegneremo 32 bit, cioè 4 byte. Nel caso dei numeri, qualunque sia il valore n scelto per rappresentare tutti gli interi tra 0 e $2^n - 1$, si deve però fare i conti con la possibilità di fare delle operazioni il cui risultato esca dalla capacità di rappresentazione scelta. Se per esempio destiniamo 4 bit ai numeri interi che vanno da 0 a 15 e sommiamo 10 con 12, il risultato 22 è maggiore del massimo numero rappresentabile; ciò porta a un errore di *overflow*, ben visibile in figura 2.11b,

legato al fatto che l'1 di riporto non trova spazio per essere collocato, poiché sarebbe necessario disporre di una quinta cella di memoria, che invece non esiste. Si osservi che per risolvere il problema non è sufficiente aumentare n, poiché per qualunque valore abbia ci saranno sempre dei numeri interi la cui somma eccede la capacità di rappresentazione. Nei computer moderni in presenza di un errore di *overflow* avviene una commutazione automatica della notazione, che passa da quella intera a quella detta *a virgola mobile* (o *floating point*), che sarà analizzata in un prossimo paragrafo.

La rappresentazione mediante complemento a 2 La notazione posizionale in base 2 ci consente di rappresentare tutti i numeri naturali tra 0 e $2^n - 1$. Resta ancora da risolvere il problema della rappresentazione dei numeri interi negativi, degli interi che escono dall'intervallo $0 \dots 2^n - 1$ e di quelli non interi (positivi o negativi che siano).
Cominciamo con gli interi negativi, che assieme ai positivi e allo 0 costituiscono l'insieme $\mathbb{Z}$. Di primo acchito potremmo ingenuamente pensare di caratterizzare il segno + o − di un numero anteponendo semplicemente 1 o 0 al numero in questione. In tal modo si otterrebbe la codifica di figura 2.12a, che consentirebbe di codificare con 4 bit tutti gli interi da -7 a $+7$. In questo modo si avrebbe il problema di una doppia rappresentazione dello zero, che non è mai opportuna poiché costringe a un doppio controllo quando si debba fare una verifica del tipo $x = 0$, per una certa variabile x. Ma il motivo che impedisce l'uso di una tale codifica è che la codifica della somma (algebrica) tra due numeri *non* corrisponde alla somma delle codifiche. Come esempio possiamo vedere la somma tra -3 e $+4$ di figura 2.12b, che produce -7 invece di $+1$. Per risolvere il problema si ricorre allora alla

0111	+7	1111	−7
0110	+6	1110	−6
0101	+5	1101	−5
0100	+4	1100	−4
0011	+3	1011	−3
0010	+2	1010	−2
0001	+1	1001	−1
0000	+0	1000	−0

(a) Rappresentazione ingenua (sbagliata) dei numeri negativi

1	0	1	1	−3	+
0	1	0	0	+4	=
1	1	1	1	−7	

(b) Errore nella somma dovuto a una rappresentazione inadeguata dei numeri negativi

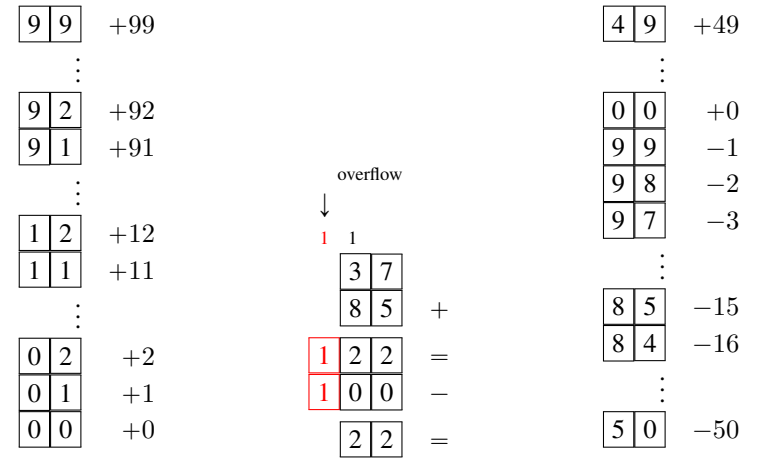
Figura 2.12: Rappresentazione errata dei numeri negativi

rappresentazione mediante *complemento a due*, che oggi è usata nella totalità dei processori, anche se in passato si era diffusa, in un primo momento, una rappresentazione simile chiamata *complemento a uno*.
Introdurremo la notazione *complemento a due* partendo da un esempio con la base decimale, con la quale abbiamo più confidenza (si tratterebbe in tal caso di un *complemento a dieci*).
Si supponga di avere a disposizione due celle decimali di memoria; ciascuna cella può contenere un numero qualunque tra 0 e 9, e dunque con esse possiamo rappresentare tutti gli interi tra 0 e $10^2 - 1 = 99$ (fig.2.13a). Supponiamo di voler fare la somma tra 37 e -15; servirebbe allora una rappresentazione coerente di -15, dove per "coerente" intendiamo che la somma algebrica delle codifiche porti alla codifica della somma. Se dobbiamo fare $37 - 15$ si può operare aggiungendo e togliendo 100 come segue

$$\begin{aligned} 37 - 15 &= 37 + (100 - 15) - 100 \\ &= 37 + 85 - 100 \\ &= 122 - 100 = 22 \end{aligned} \tag{2.3}$$

Il risultato non cambia, cioè 22, ma questo approccio ci fa capire che per ottenere il risultato corretto bisogna togliere 100 alla somma tra 37 e il complemento a 100 di 15, che è 85 (complemento a 100 significa in questo caso

"ciò che manca a 15 per arrivare a 100"). Se eseguiamo l'operazione sottoponendola ai vincoli di memoria imposti, possiamo osservare (fig.2.13b) che sommando 37 a 85 su una memoria con due sole celle si perde il riporto di 1 per *overflow*, e questa perdita corrisponde a una sottrazione automatica di 100 dal risultato finale, che porta a un risultato corretto. Poiché la somma di 37 e 85 è pari a 22 (in *overflow*), possiamo lecitamente *interpretare il numero* 85 *come una rappresentazione di* -15. In tal modo la rappresentazione di $-x$ diventa quella di $10^2 - x$, e quindi -1 è codificato dal 99, -2 dal 98 ecc. Il più grande numero positivo che si può codificare è $+49$, mentre il più grande numero negativo è -50 (fig.2.13c). La notazione basata sul complemento (a dieci, in questo caso) gode del



(a) Rappresentazione degli interi da 00 a 99

(b) L'errore di *overflow* consente di sottrarre 100 in modo automatico

(c) Rappresentazione mediante complemento a 10

Figura 2.13: Rappresentazione decimale tradizionale e a complemento a 10 con due celle di memoria

grosso vantaggio che le somme e le sottrazioni si eseguono sempre come somma (algebrica) delle codifiche, con un unico tipo di circuiteria. Si ha inoltre un'unica rappresentazione dello zero. La possibilità di rappresentare i numeri negativi porta a un bilanciamento attorno allo zero della capacità di rappresentazione, che passa da $0 \ldots 99$ a $-50 \ldots +49$.

Se ora passiamo al caso binario cambia la base, ma non il ragionamento. Supponiamo di avere 4 bit a disposizione per la codifica. Invece che rappresentare gli interi tra 0 e 15 vogliamo centrare attorno allo 0 l'intervallo di rappresentazione, in modo da disporre anche di una codifica per i numeri negativi. Con un ragionamento simile a quello fatto prima possiamo notare che, p.es., la somma algebrica tra 5 e -2 può essere scritta aggiungendo e togliendo 2^4 (in generale b^n con b base e n numero di celle di memoria)

$$\begin{aligned} 5-2 &= 5+(16-2)-16 \\ &= 5+14-16 \\ &= 19-16=3 \end{aligned} \tag{2.4}$$

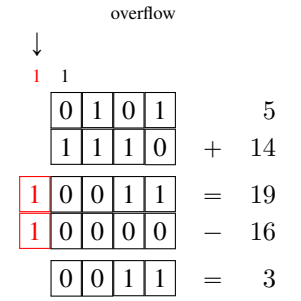
In questo modo la codifica di -2 diventa quella del complemento a $2^4 = 16$, cioè 14; quest'ultimo sommato con 5 porta a 19, al quale viene tolto 16 in modo automatico dall'overflow su 4 bit (fig.2.14b). La codifica degli interi tra 0 e 15 di figura 2.14a viene allora trasformata nella codifica tra -8 e $+7$ di figura 2.14c. Se disponiamo di n bit, l'intervallo di rappresentazione passa da $[0, 2^n - 1]$ per la codifica dei soli numeri positivi a

$$[-2^{n-1}, +2^{n-1} - 1]$$

per la codifica dei numeri interi positivi e negativi, il che equivale a centrare l'intervallo di rappresentazione sullo 0 (si veda figura 2.15).

1	1	1	1	+15
1	1	1	0	+14
1	1	0	1	+13
1	1	0	0	+12
1	0	1	1	+11
1	0	1	0	+10
1	0	0	1	+9
1	0	0	0	+8
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0

(a) Rappresentazione degli interi da 0 a 15



(b) L'errore di *overflow* consente di sottrarre 16 in modo automatico

0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	1	−1
1	1	1	0	−2
1	1	0	1	−3
1	1	0	0	−4
1	0	1	1	−5
1	0	1	0	−6
1	0	0	1	−7
1	0	0	0	−8

(c) Rappresentazione mediante complemento a 2 con 4 bit

Figura 2.14: Rappresentazione binaria tradizionale e a complemento a 2 con 4 celle di memoria

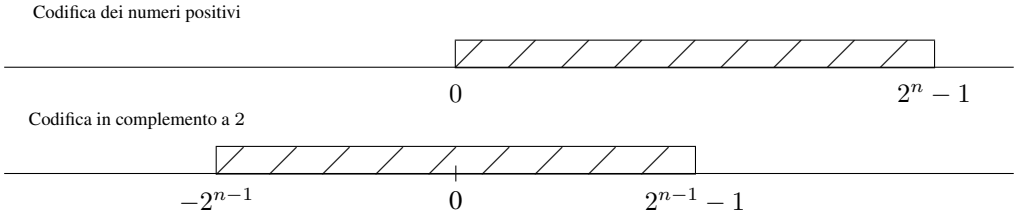


Figura 2.15: Intervallo di rappresentazione degli interi nella notazione mediante complemento a 2 usando n celle binarie

La regola che deriva è la seguente

$$\text{codifico } \pm x \qquad \text{come} \qquad 2^n \pm x \qquad \text{letto con } n \text{ bit in } \textit{overflow}$$

e quindi -3 ha la codifica di $2^4 - 3 = 13$, cioè 1101, mentre $+3$ ha la codifica di $2^4 + 3 = 19$ (10011), che letto con 4 bit in overflow corrisponde a 0011. In pratica la codifica di un numero negativo $-x$ si ottiene complementando la rappresentazione binaria di $+x$ e poi sommando 1. Ciò deriva dalla circostanza che per ogni x, se sommiamo la sua rappresentazione in binario con quella del proprio complemento $\overline{x}$, ottenuta scrivendo 0 al posto di 1 e viceversa, si ottiene

$$x + \overline{x} = 2^n - 1 \qquad \text{cioè} \qquad \overline{x} + 1 = 2^n - x \tag{2.5}$$

Vale la pena osservare la genialità della rappresentazione mediante complemento a 2, che sfrutta un difetto del sistema (l'*overflow*) per realizzare una efficace codifica dei numeri negativi.

Nei computer attuali il numero di bit che si usano per i numeri interi è 32 o 64; questa scelta consente di rappresentare i seguenti intervalli:

$$\begin{array}{lcccll}
32 \text{ bit} & \Rightarrow & [0, 2^{32}-1] & \Rightarrow & [0, & 4.294.967.295] \text{ positivi} \\
32 \text{ bit} & \Rightarrow & [-2^{31}, +2^{31}-1] & \Rightarrow & [-2.147.483.648, & 2.147.483.647] \text{ complemento a 2} \\
64 \text{ bit} & \Rightarrow & [0, 2^{64}-1] & \Rightarrow & [0, & 1,84467E+19] \text{ positivi} \\
64 \text{ bit} & \Rightarrow & [-2^{63}, +2^{63}-1] & \Rightarrow & [-9,22337E+18, & +9,22337E+18] \text{ complemento a 2}
\end{array}$$

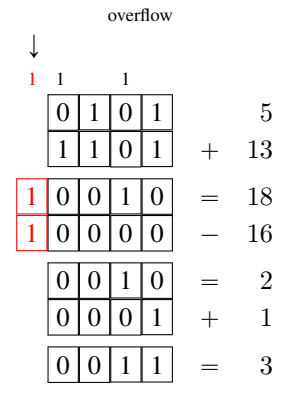
Si può notare che 32 bit non sono sufficienti in pratica, poiché si arriva a poco oltre i due miliardi nella notazione in complemento a 2, che è un numero intero che può capitare di dover trattare.

La rappresentazione mediante complemento a 1 Per completezza, facciamo ora un breve cenno alla notazione mediante complemento a 1; anche se è oramai divenuta obsoleta, tale notazione fu usata in macchine che hanno segnato la storia recente dell'informatica (CDC 6600, Univac 1100/2200). La notazione mediante complemento a 1 è una variante di quella a complemento a 2.
Riprendiamo le equazioni (2.4) e (2.5) e riscriviamole aggiungendo e togliendo b^n-1, con $b = 10$ e $b = 2$ rispettivamente. Si ottiene

$$\begin{array}{rcl}
37-15 &=& 37+(99-15)-99 \\
&=& 37+84-99 \\
&=& 121-99 \\
&=& 121-(100-1) \\
&=& 121-100+1 \\
&=& 21+1=22
\end{array}
\qquad
\begin{array}{rcl}
5-2 &=& 5+(15-2)-15 \\
&=& 5+13-15 \\
&=& 18-15 \\
&=& 18-(16-1) \\
&=& 18-16+1 \\
&=& 2+1=3
\end{array}
\tag{2.6}$$

1	1	1	1	+15
1	1	1	0	+14
1	1	0	1	+13
1	1	0	0	+12
1	0	1	1	+11
1	0	1	0	+10
1	0	0	1	+9
1	0	0	0	+8
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0

(a) Rappresentazione degli interi da 0 a 15



(b) L'errore di *overflow* consente di sottrarre 16 in modo automatico; poi bisogna sommare 1 per bilanciare l'equazione

0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	1	−0
1	1	1	0	−1
1	1	0	1	−2
1	1	0	0	−3
1	0	1	1	−4
1	0	1	0	−5
1	0	0	1	−6
1	0	0	0	−7

(c) Rappresentazione mediante complemento a 1 con 4 bit

Figura 2.16: Rappresentazione binaria tradizionale e a complemento a 1 con 4 celle di memoria

Si noti che ora la rappresentazione di $-x$ è quella di $(2^n–1)-x$; in binario ciò corrisponde a fare la negazione (il complemento a 1) di ogni cifra binaria di x; in questo modo la rappresentazione di -2 è il complemento a 1

di 0010, cioè 1101; la tabella di figura 2.16c riporta la codifica di tutti gli interi compresi tra -7 e $+7$. Nel fare la differenza riportata nell'equazione (2.6) ora dobbiamo sottrarre $2^4 - 1 = 15$; il valore 2^4 viene sottratto automaticamente dall'overflow, mentre il valore 1 deve essere sommato alla fine, per bilanciare l'equazione. Si osservi tuttavia che, se il valore della differenza è negativo, allora non è necessario sommare 1:

$$\begin{aligned} 2-5 &= 2+(15-5)-15 \\ &= 2+10-15 \\ &= 12-15 \\ &= -3 \end{aligned} \qquad \begin{array}{cccc} & & \overset{1}{} & \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 \end{array} \quad \begin{array}{c} 2 \\ + \ 10 \\ = \ 12 \end{array} \tag{2.7}$$

Ciò accade perché in tal caso la stringa binaria che si ottiene (1100), e che codifica il numero 12, rappresenta già la notazione in complemento a 1 del risultato (-3).
Questo tipo di codifica soffre di due difetti principali: il primo è la circostanza che la differenza va eseguita in modo diverso a seconda che il risultato sia positivo o negativo; il secondo, non meno grave, è dato dalla doppia rappresentazione dello 0, associato tanto alla stringa 0000 che alla 1111. Questo è il motivo principale del suo disuso.

La notazione a virgola mobile o *floating point*

Risolto il problema della rappresentazione dei numeri interi, bisogna ora affrontare quelli dei numeri non interi. Come noto essi sono i *numeri razionali* $\mathbb{Q}$, che si possono rappresentare come frazione a/b con a e b interi, e la loro estensione dei *numeri reali* $\mathbb{R}$. Questi ultimi contengono $\mathbb{Q}$ e $\mathbb{Z}$ come sottinsieme, ma contengono anche i numeri *irrazionali algebrici* (come la radice quadrata di 2, che non è esprimibile nella forma a/b) e i *numeri irrazionali trascendenti* (come π ed e, che non sono soluzione di alcuna equazione polinomiale a coefficienti razionali). Poiché i numeri reali hanno in genere un'espansione decimale infinita, per descrivere un singolo reale all'interno di un calcolatore sarebbe necessaria una quantità infinita di memoria. I reali non sono dunque gestibili a livello informatico, se non che mediante un *troncamento* (o un *arrotondamento*) delle cifre che ci consenta di trattarli in modo approssimato. Per esempio se dobbiamo lavorare col numero *Pi greco*

$$\pi = 3,14159265358979323846264338327950288419716939937510582097494 4\ldots$$

nel quale i tre punti esprimono il fatto che ci sono altre infinite cifre, per poterlo gestire è necessario effettuare un troncamento con un certo numero di cifre. Se scegliamo p.es. 10 cifre dopo la virgola, la rappresentazione approssimata di *Pi greco* diventa

$$\pi = 3,1415926535$$

che contenendo un numero finito di cifre decimali potrà essere gestita da una memoria di un computer. È ovvio che maggiore è il numero di cifre che usiamo e migliore sarà la precisione dei nostri calcoli. A questo punto il numero $3,1415926535$ può facilmente essere espresso nella forma a/b di un numero razionale, poiché possiamo scriverlo come

$$3,1415926535 = 31.415.926.535/10.000.000.000 = 31415926535 \cdot 10^{-10} \tag{2.8}$$

dove nell'ultima uguaglianza abbiamo usato la *notazione esponenziale*, detta anche *notazione scientifica* quando la base è 10. Se riusciamo a rappresentare in modo efficace i numeri razionali, potremo dunque lavorare anche con le approssimazioni dei reali. Come si può notare dalla rappresentazione (2.8), per identificare in modo univoco il numero $3,1415926535$ basta disporre dei due numeri interi della notazione esponenziale, che sono rispettivamente 31415926535 e -10, e concordare la base dell'esponente (in questo caso 10). Con questa convenzione si può scrivere un generico numero N come

$$N = \pm\, m \cdot 10^{\pm e}$$

Al numero m si dà il nome di *mantissa*, mentre e è chiamato *esponente*.
La notazione esponenziale è estremamente flessibile, poiché al prezzo di una piccola perdita di precisione, data dal

fatto che la mantissa ha un numero prefissato di cifre che potrebbe essere inferiore al numero di cifre significative che si vogliono rappresentare, si possono gestire numeri molto grandi o molto piccoli, positivi o negativi che siano. Per esempio

$$\begin{aligned} +12\,300 &= +1,23 \cdot 10^{+4} = +123 \cdot 10^{+2} \\ -4\,500\,000\,000 &= -4,5 \cdot 10^{+9} = -45 \cdot 10^{+8} \\ -0,000\,67 &= -6,7 \cdot 10^{-4} = -67 \cdot 10^{-5} \\ +0,000\,000\,089 &= +8,9 \cdot 10^{-8} = +89 \cdot 10^{-9} \end{aligned}$$

Poiché sappiamo già come rappresentare un intero, possiamo codificare un numero razionale usando m, e e la rispettiva coppia di segni. Dovremo evidentemente decidere quanti bit attribuire alla mantissa, quanti all'esponente e poi serviranno due bit per il segno. In questo modo la rappresentazione diventa

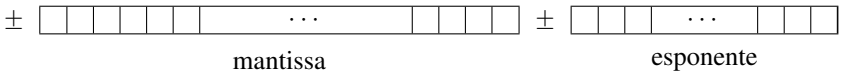


Figura 2.17: Notazione *floating point*

e viene chiamata notazione *a virgola mobile* o *floating point* (con la convenzione che la base sia 2).
Sempre ragionando in base 10, che ci è più familiare, e supponendo di attribuire p.es. 5 celle di memoria alla mantissa e 2 all'esponente, si voglia rappresentare il numero $23,89729$. Per far emergere 5 cifre per la mantissa bisogna spostare la virgola di 3 posizioni a destra e poi moltiplicare per 10^{-3}, ottenendo $23897,29 \cdot 10^{-3}$; ma nella rappresentazione le due cifre 29 dopo la virgola devono essere sacrificate, perdendo in precisione (fig.2.18). Poiché

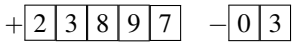


Figura 2.18: Rappresentazione in *floating point* del numero 23,89729 nella quale si perdono le ultime due cifre significative

il 29 è andato perso, è ovvio che al suo posto avrebbe potuto esserci qualunque altra coppia di interi compresi tra 00 e 99. Ecco allora che tutti i numeri che stanno a sinistra della figura 2.19 hanno la stessa rappresentazione di destra.

Con la stessa notazione basata su 5 celle per la mantissa e 2 per l'esponente, si possono rappresentare numeri

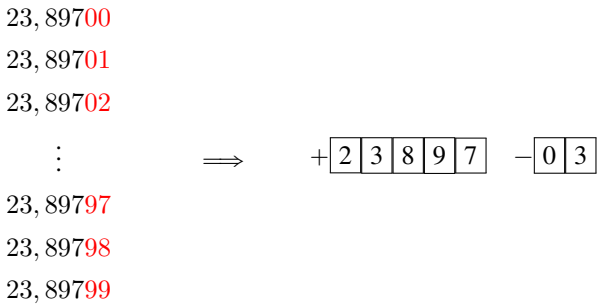


Figura 2.19: Tutti i numeri di sinistra hanno la stessa rappresentazione *floating point* di destra

con una dinamica enorme, da $+99999 \cdot 10^{+99}$ a $+00001 \cdot 10^{-99}$ per i numeri positivi, e da $-00001 \cdot 10^{-99}$ a $-99999 \cdot 10^{+99}$ per quelli negativi, a scapito della piccola perdita di precisione dopo la quinta cifra significativa. Si osservi inoltre che per l'aritmetica dei numeri basati sulla notazione a virgola mobile non valgono le leggi associative e distributive della somma e moltiplicazione; per esempio $x+(y+z) \neq (x+y)+z$, e dunque cambiando *l'ordine* con il quale vengono fatte le operazioni può cambiare il risultato. Sempre riprendendo l'esempio di prima

si ha

$$1 \cdot 10^{-4} + (23897 \cdot 10^{-3} - 23897 \cdot 10^{-3}) = 1 \cdot 10^{-4} \quad \text{ma} \quad (1 \cdot 10^{-4} + 23897 \cdot 10^{-3}) - 23897 \cdot 10^{-3} = 0$$

In generale possiamo dire che la notazione a virgola mobile offre una straordinaria fessibilità, legata alla possibilità di rappresentare numeri molto grandi e molto piccoli, a scapito di una piccola perdita di precisione.
Per tornare al caso binario possiamo dire che ci sono sostanzialmente due standard che hanno preso piede nell'architettura dei calcolatori denominati rispettivamente IEEE 754 a 32 e 64 bit. In entrambi la struttura di base è costituita dal bit del segno del numero, seguito dall'esponente espresso in una notazione *traslata* (che rappresenta tanto esponenti positivi che negativi), seguito infine dalla dalla mantissa. Cominciamo con la IEEE 754-32, denominata anche *single precision*, di cui vediamo la struttura in figura 2.20. Come si può notare c'è un segno,

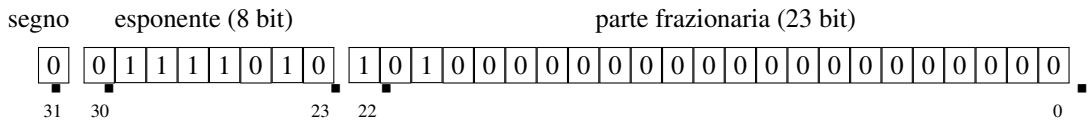


Figura 2.20: Rappresentazione *floating point* IEEE 754-32 o *single precision*

seguito da 8 bit per l'esponente e dalla mantissa che in questo caso rappresenta 23 bit della *parte frazionaria* del numero $1, b_{-1}b_{-2}\ldots b_{-23}$ (con i b_{-i} espressi in base 2). Il valore 1 prima della virgola viene sottinteso, e quindi non si riporta nella notazione risparmiando un bit. Tuttavia esso viene eliminato quando nell'esponente ci sono solamente zeri.
Per l'esponente si usa una notazione speciale, denominata *a eccesso* 127, nella quale il numero letto nella notazione posizionale in binario rappresenta il numero da codificare + 127, e dunque per dedurre il numero codificato dobbiamo sottrarre 127 al numero letto. La tabella di figura 2.21 ci fa vedere la conversione.

Valore binario	Interpretazione a eccesso 127	Lettura in notazione posizionale
11111111	+128	255
11111110	+127	254
11111101	+126	253
⋮		
10000001	+2	129
10000000	+1	128
01111111	0	127
01111110	−1	126
⋮		
00000010	−125	2
00000001	−126	1
00000000	−127	0

Figura 2.21: Rappresentazione mediante *eccesso a* 127

Tenuto conto di ciò il numero che deriva dalla notazione IEEE 754-32 *single precision* è nella forma

$$(-1)^{\text{segno}}\,(1, b_{-1}b_{-2}\ldots b_{-23}) \cdot 2^{e-127} = (-1)^{\text{segno}} \left(1 + \sum_{i=1}^{23} b_{23-i}\, 2^{-i}\right) \cdot 2^{e-127}$$

Vediamo allora, come esempio, il numero che corrisponde alla codifica di figura 2.20, cioè

0 01111010 10100000000000000000000

Il segno vale 0. L'esponente 01111010 espresso in notazione posizionale vale $2^6 + 2^5 + 2^4 + 2^3 + 2^2 = 122 = e$, che sottraendo 127 porta a -5. Per la parte frazionaria si ha invece $1 + 2^{-1} + 2^{-3} = 1,625$. Si ottiene allora

$$(-1)^{\text{segno}} \left(1 + \sum_{i=1}^{23} b_{23-i}\, 2^{-i}\right) \cdot 2^{e-127} = (-1)^0\,(1,625) \cdot 2^{-5} = 5,0781$$

L'altra notazione *floating point* di tipo standard, la IEEE 754 a 64 bit detta *double precision*, è del tutto simile a quella a 32 bit, con l'unica differenza che riserva 11 bit per l'esponente e 52 bit per la parte frazionaria (fig.2.22). Per la lettura dell'esponente si fa sempre riferimento alla notazione a eccesso, solo che in questo caso l'eccesso

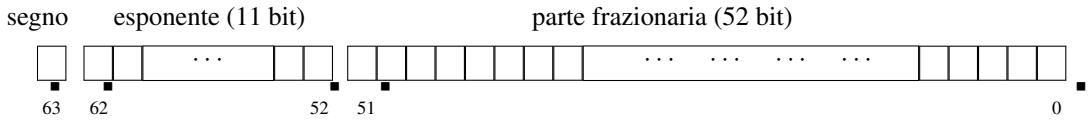


Figura 2.22: Rappresentazione *floating point* IEEE 754-64 o *double precision*

vale 1023 poiché l'esponente ha 11 bit invece che 8. Con gli stessi simboli usati per il caso precedente si ottiene allora

$$(-1)^{\text{segno}}\,(1, b_{-1}b_{-2}\ldots b_{-52}) \cdot 2^{e-1023} = (-1)^{\text{segno}} \left(1 + \sum_{i=1}^{52} b_{52-i}\, 2^{-i}\right) \cdot 2^{e-1023}$$

Sulla base di quanto visto finora possiamo usare i 64 bit (o i 32 bit) per la rappresentazione dei numeri o nella notazione a complemento a 2 per codificare i numeri interi positivi e negativi, oppure i numeri non interi usando la notazione a virgola mobile. Nel caso in cui, lavorando con gli interi, si dovesse superare la soglia di rappresentazione si avrà una commutazione automatica alla notazione *floating point*, perdendo eventualmente un po' di precisione.

La rappresentazione fisica del *bit*

Prima di passare alla descrizione delle modalità che ci consentono di codificare i segnali analogici e le immagini, facciamo una breve digressione sul problema della rappresentazione fisica del *bit*. Come già spiegato precedentemente tutte le informazioni del mondo esterno devono essere codificate in stringhe binarie, cioè stringhe su un alfabeto di due elementi. Ogni singola cifra binaria, che può essere 0 o 1, viene chiamata *bit*, parola che deriva dalla contrazione di *Binary digIT*.
Il *bit* è la più piccola quantità d'informazione possibile, cioè quella che ci consente di distinguere tra due due stati alternativi ed equiprobabili.
Quando per esempio lanciamo una moneta (non truccata), per la quale la probabilità a priori di ottenere testa o croce è pari a $1/2$, l'informazione che ne ricaviamo è di 1 *bit*.

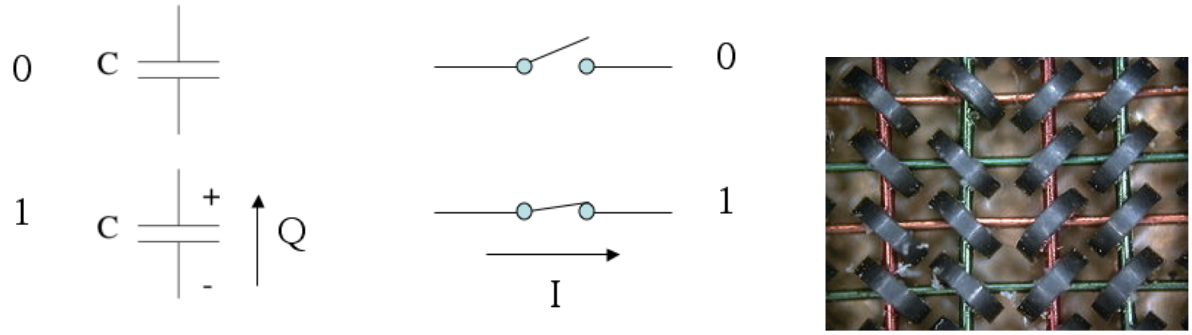
Per quanto riguarda i multipli del *bit* c'è da dire che essi non sono praticamente usati, poiché in informatica si tende a usare il *byte* come unità d'informazione. Ciò deriva dal fatto che il *byte* è solitamente la più piccola quantità di memoria *indirizzabile* all'interno di un computer, quella associata a una lettera nel codice ASCII. Purtroppo, però, questo non è l'unica specificità relativa ai multipli che si usano in informatica, poiché esiste una più seria ambiguità legata al valore dei prefissi *kilo, mega, tera*, ecc., introdotti nel *Sistema Internazionale* come multipli in base 10 e usati in Informatica intendendo multipli in base 2. Di conseguenza quando si dice $kilo$ si pensa a un fattore 10^3 nel *Sistema Internazionale*, ma a un fattore $2^{10} = 1024$ in ambito informatico. La necessità di usare la base 2 deriva dal fatto che con n bit si indirizzano 2^n celle di memoria, e quindi le quantità di memoria è sempre espressa come potenza di 2. Poiché $1024 \approx 1000$, si usa impropriamente il multiplo *kilo* per indicare 2^{10}, mentre i prefissi *Mega* ($2^{20} = 1.048.576$) e *Giga* ($2^{30} = 1.073.741.824$) vengono usati come approssimazione di milione e miliardo. Gli organismi internazionali di standardizzazione hanno tentato di imporre una denominazione non ambigua (si veda la colonna di sinistra della tabella di figura 2.23), basata sull'introduzione della parola *binary* nella struttura del nome (*kilo binary byte* = *kibibyte* e così via), ma essa non è però entrata nell'uso comune. Si

noti che la differenza percentuale tra valore effettivo e multiplo in base 10 del Sistema Internazionale porta a degli errori che sono percentualmente crescenti con l'ordine di grandezza (si veda l'ultima colonna della tabella 2.23). Si è detto che l'esistenza del teorema di campionamento ha decretato il trapasso dalla tecnologia analogica a quella

Prefissi binari				Prefissi SI			
Fattore	Simbolo	Nome		Simbolo	Nome	Fattore	Errore %
2^{10}	KiB	$kibibyte$	$\approx$	kB	$kilobyte$	10^3	$+2,4$
2^{20}	MiB	$mebibyte$	$\approx$	MB	$megabyte$	10^6	$+4,9$
2^{30}	GiB	$gibibyte$	$\approx$	GB	$gigabyte$	10^9	$+7,4$
2^{40}	TiB	$tebibyte$	$\approx$	TB	$terabyte$	10^{12}	$+10,0$
2^{50}	PiB	$pebibyte$	$\approx$	PB	$petabyte$	10^{15}	$+12,6$
2^{60}	EiB	$exbibyte$	$\approx$	EB	$exabyte$	10^{18}	$+15,3$
2^{70}	ZiB	$zibibyte$	$\approx$	ZB	$zettabyte$	10^{21}	$+18,1$
2^{80}	YiB	$yobibyte$	$\approx$	YB	$yottabyte$	10^{24}	$+20,9$

Figura 2.23: Multipli del byte usando prefissi binari (praticamente inutilizzati) e prefissi del *Sistema Internazionale*, che esprimono però un valore approssimato

digitale, e che l'esistenza della logica Booleana, assieme ad alcune altre questioni di carattere tecnico, ha fatto prevalere i circuiti binari rispetto a qualunque altra tipologia discreta (ternario, esadecimale, ecc; cfr. sez. 2.2)). Ora resta da capire come prendono corpo i *bit* all'interno dei circuiti elettronici che costituiscono il processore e le memorie dei *computer*.
In generale un *bit* può esser associato ai livelli di una grandezza fisica di riferimento, che per i dispositivi moderni può essere tipicamente una tensione, una corrente, una carica elettrica o la presenza di una polarizzazione magnetica. In figura 2.24 possiamo vedere alcuni esempi; nel condensatore di figura 2.24a il valore logico 1 è associato alla presenza di una carica elettrica Q tra le armature dello stesso condensatore, il valore logico 0 alla sua assenza, mentre nell'interruttore di fig. 3.1 i due stati logici 0 e 1 sono rappresentati dall'interruttore aperto (non c'è flusso di corrente) e dall'interruttore chiuso (scorre una corrente I). Un discorso simile vale per la polarizzazione che si ha quando un nucleo toroidale di *ferrite* viene polarizzato N-S (Nord-Sud) o S-N, a seconda del verso della corrente che scorre nel filo che attraversa il nucleo. La figura 2.24c) mostra in particolare la tecnica usata nei calcolatori degli anni '50 -'60 per assemblare i nuclei di ferrite in modo da costituire blocchi estesi di memoria; nella memoria della figura sono presenti 16 nuclei associati a 16 bit.



(a) Condensatore nei due stati logici possibili, carico e scarico

(b) Interruttore nei due stati logici possibili, acceso e spento

(c) Nuclei di ferrite di una memoria in uso negli anni '50-'60

Figura 2.24: Rappresentazione fisica del *bit* usando diverse grandezze elettriche e magnetiche

Attualmente si tende a usare un transistor come supporto fisico del *bit*, soprattutto nella sua variante MOS-FET (*Metal Oxide Semiconductor- Field Effect Transistor* o transistor *a effetto di campo*). Se osserviamo la figura 2.6,

notiamo che si può prendere in considerazione la tensione v tra collettore ed emettitore di un transistor e associare il valore logico 1 a un valore elevato di v (*logica positiva*) e un valore logico 0 a un valore basso della stessa tensione. Nei circuiti integrati delle famiglie logiche TTL (*Transistor-Transistor-Logic*) la tensione v di uscita associata allo stato 1 deve essere maggiore di $2,6$ V, mentre la tensione dello stato 0 deve essere minore di $0,4$ V (fig. 2.25). Un secondo metodo per realizzare fisicamente un *bit*, usato soprattutto per le memorie e che deriva dal

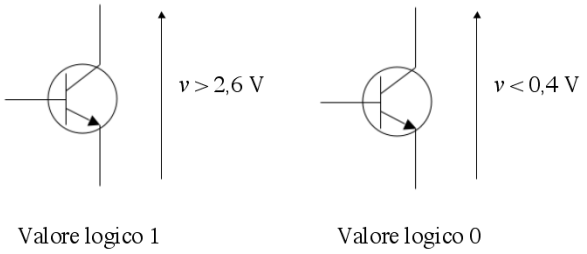
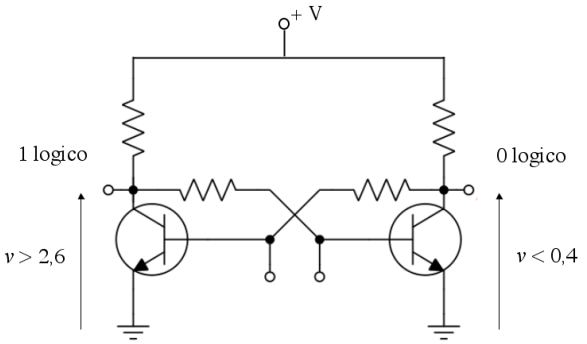
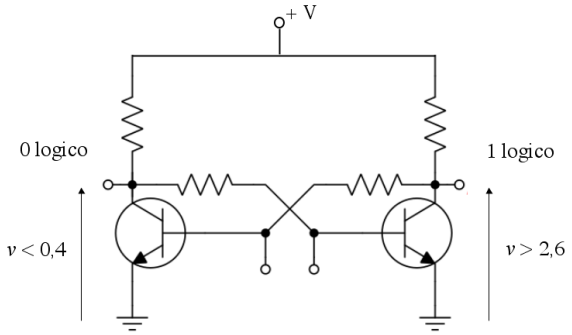


Figura 2.25: Valore logico 1 e 0 per la famiglia di circuiti integrati TTL

circuito di fig. 2.25, è quello di connettere l'uscita di un transistor in interdizione con l'ingresso di un transistor in conduzione, realizzando il cosiddetto *Flip-Flop* (si veda figura 2.26) Il circuito che si ottiene è *bistabile*, nel senso



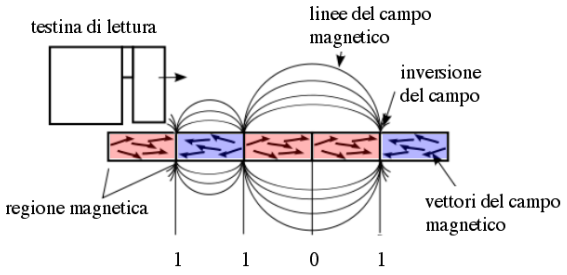
(a) Il transistor di sinistra del *Flip-Flop* è interdetto; ciò porta in conduzione il transistor di destra, che fornisce uscita logica 0



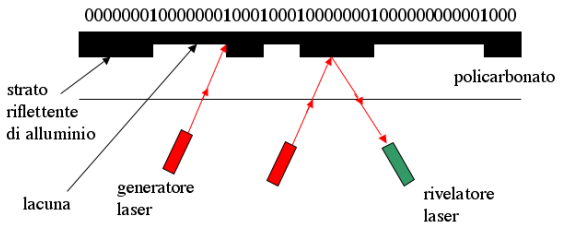
(b) Il transistor di sinistra del *Flip-Flop* è in conduzione; ciò porta in interdizione il transistor di destra, che fornisce uscita logica 1

Figura 2.26: Circuito bistabile del *Flip-Flop*

che esso può stare indifferentemente e stabilmente in uno o nell'altro dei due stati rappresentati in figura 2.26a e 2.26b. Il funzionamento è basato sul fatto che, quando il transistor di sinistra è interdetto (non passa corrente nel circuito di collettore), allora la sua tensione di collettore è alta ($v > 2,6$ V); ciò polarizza la base del transistor di destra, che entra in piena conduzione, facendo collassare a un valore basso ($v < 0,4$ V) la sua tensione di uscita (fig. 2.26a), che corrisponde a uno 0 logico. I ruoli dei due transistor si scambiano quanto la tensione di collettore del transistor di destra viene portata (con un impulso) a un valore ($v > 2,6$ V) (fig. 2.26b).
I circuiti a *Flip-Flop* possono essere realizzati anche con i MOS-FET prima citati, che danno migliori caratteristiche per quanto riguarda le potenze dissipate per il funzionamento, e gli stessi MOS-FET vengono impiegati, in una versione speciale, anche come elemento per le memorie *flash*, che sono memorie elettroniche di tipo permanente.
Altri sostrati fisici usati per memorizzare i *bit* sono quelli basati sul sottile strato di magnetizzazione che ricopre la superficie dei dischi rigidi (*Hard Disk Drive*, vedi figura 2.27a), che viene suddiviso in piccole aree magnetizzate in senso S-N (area rosa) o N-S (area azzurra). La transizione tra due aree diverse viene associata a un 1 mentre il mantenimento della stessa area corrisponde a uno 0.
Un altro tipo di tecnologia usata per memorizzare i *bit* è quella laser dei dischi ottici (fig.2.27b). Su un disco di policarbonato viene steso uno strato sottile di alluminio, che se lasciato integro è in grado di riflettere con alta efficienza un raggio laser; la superficie di alluminio viene invece forata in corrispondenza delle zone dove la luce laser



(a) Bit associati al sostrato ferromagnetico di un disco rigido



(b) Bit associati alla variazione di riflettenza di una superficie colpita da un raggio laser

Figura 2.27: Alcuni metodi per memorizzare bit fisici su *Hard Disk* e *Compact Disk*

non dev'essere riflessa. Anche in questa tecnologia si ha un 1 in corrispondenza di una variazione, che riguarda in questo caso la riflettenza della superficie.
Le tecniche di rappresentazione fisica dei bit che abbiamo analizzato finora riguardano la *memorizzazione* degli stessi su appositi supporti, che coincidono quasi sempre con dei dispositivi a semiconduttore. Quando si deve trasferire un pacchetto di bit all'interno della scheda madre del computer, oppure tra *computer* connessi in rete, si pone il problema di come organizzare tale trasmissione. Esistono a tal riguardo due modalità diverse, chiamate rispettivamente *trasmissione seriale* e *trasmissione parallela*, illustrate in figura 2.28 con riferimento alla codifica del numero 7. Nella trasmissione seriale i singoli bit della rappresentazione posizionale del numero vengono

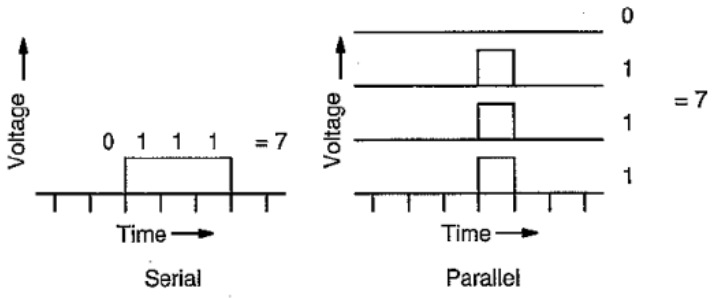
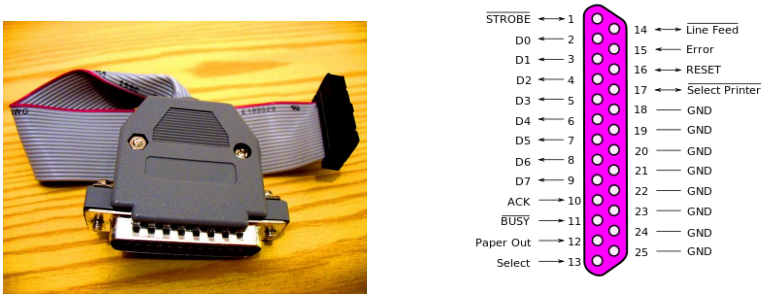


Figura 2.28: Trasmissione seriale e trasmissione parallela dei bit che codificano il numero 7

dislocati lungo l'asse temporale, nel senso che sono spediti uno dopo l'altro su una singola linea. Al contrario la trasmissione parallela prevede un numero di connessioni pari al numero di bit da trasmettere; i bit vengono a questo punto trasmessi contemporaneamente. È ovvio che, a parità di frequenza di sincronizzazione del sistema (chiamata in gergo *frequenza di clock*), il sistema parallelo a n bit offre una velocità di trasferimento n volte superiore al sistema seriale. In cambio esso è più costoso, poiché il cavo di interconnessione deve contenere $n + 1$ conduttori al posto dei 2 sufficienti per il caso seriale. I dispositivi fisici di connessione (parallela) vengono solitamente chiamati *bus*, e sono costituiti da un cavo con molti conduttori in parallelo. Nella figura 2.29a si può osservare un cavo parallelo con il relativo connettore (maschio), mentre nella figura e 2.29b viene rappresentata la struttura dei collegamenti standard per una *porta parallela*, cioè un connettore (femmina) che di solito si trova connesso con lo *chassis* del computer.

2.3.3 La codifica dei segnali analogici

Nella sezione 2.2 abbiamo sottolineato il fatto che l'informazione può nascere tanto in forma discreta (p.es. quella associata all'alfabeto di una tastiera) quanto in forma analogica (p.es. il suono captato da un microfono



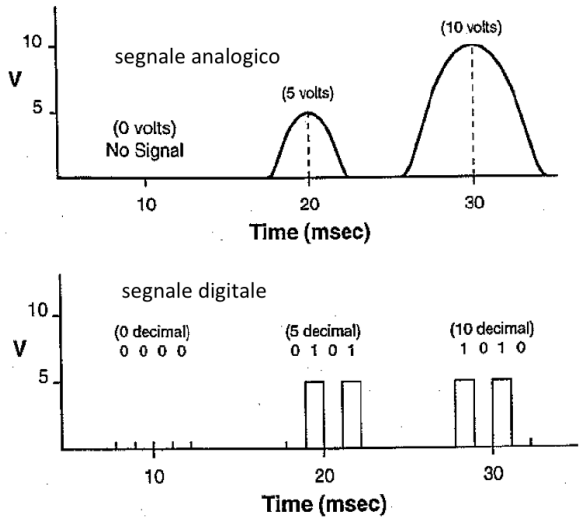
(a) Connettore (maschio) e relativo cavo parallelo

(b) Connessione circuitale di una porta parallela (femmina) associata a un *bus* di 8 bit (piedini 2-7)

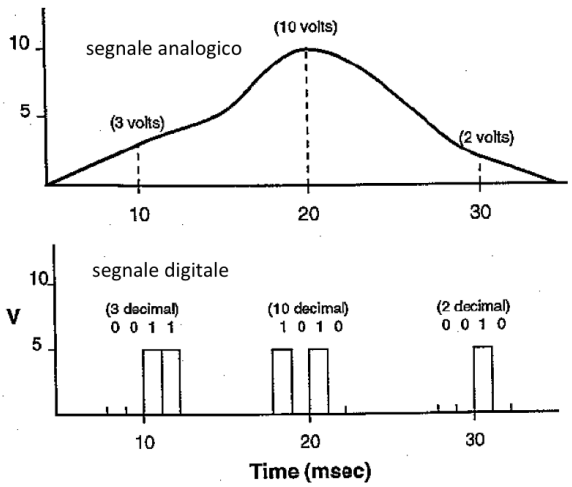
Figura 2.29: Connessione parallela

in una sessione di *Skype*). Se l'informazione è già in forma discreta abbiamo l'unico problema di codificarla in binario, p.es. usando una tabella simile a quella del codice ASCII. Se invece essa assume la forma di un segnale analogico, bisogna porsi il problema di capire se la codifica in binario sia un'operazione effettivamente realizzabile. L'informazione associata a un segnale analogico è infatti potenzialmente infinita, poiché è necessario specificare, istante per istante su una scala continua dei tempi, il valore della grandezza analizzata. Poiché tale valore è un numero reale, anche per identificare un solo punto della curva sarebbe in linea di principio necessario avere a disposizione una quantità infinita di memoria. Ne consegue che dovremo ricorrere a una rappresentazione approssimata del valore basata sulla notazione *floating point*.

Nella gestione dei segnali analogici ci sono essenzialmente due situazioni ricorrenti. Nella prima un segnale di tipo impulsivo viene usato per rappresentare un singolo valore numerico, che potrebbe derivare per esempio dalla lettura di una grandezza fisica; è questo il caso di figura 2.30a, nella quale l'impulso in uscita da un tubo fotomoltiplicatore (rilevatore di luce ad altissima sensibilità) viene valutato nella sua ampiezza in corrispondenza del picco di tensione. In questo caso il segnale analogico funge solo da "supporto" all'informazione relativa all'ampiezza. Nel secondo esempio di figura 2.30b si vede invece un segnale analogico in uscita da una video camera



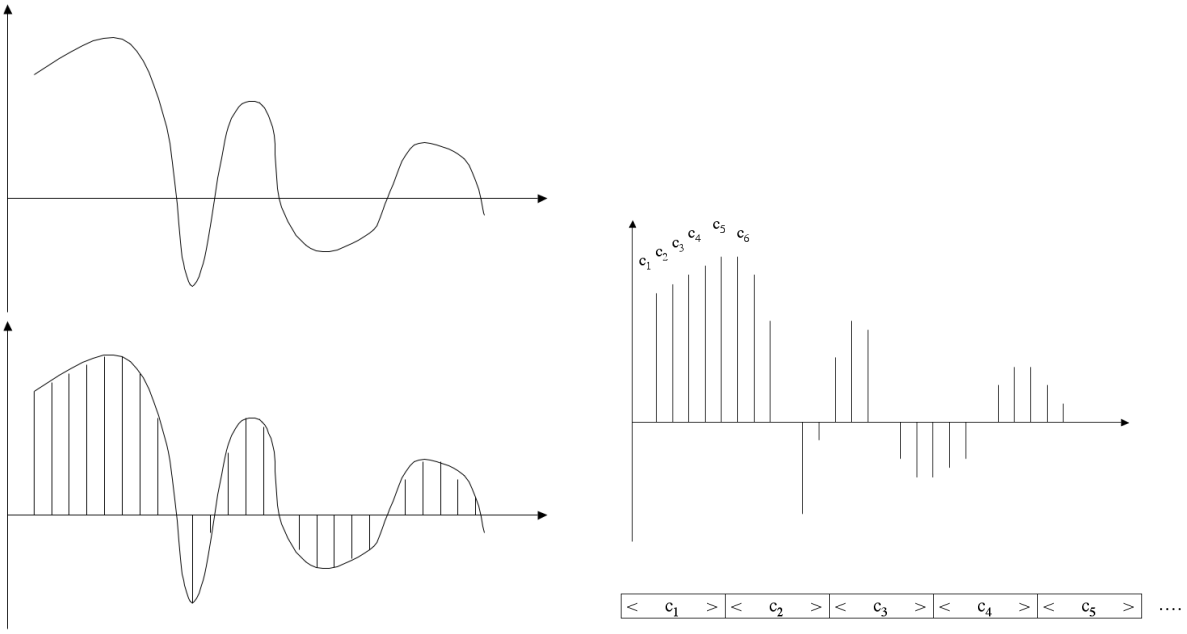
(a) Segnale analogico in uscita da un tubo fotomoltiplicatore connesso con uno scintillatore



(b) Segnale analogico in uscita da una video camera in un sistema a fluoroscopia

Figura 2.30: Esempi di segnali analogici generati da apparecchiature elettromedicali con le corrispondenti codifiche binarie dei valori misurati a intervalli di tempo costanti

in un sistema a fluoroscopia, che varia in modo continuo nel tempo, seguendo le variazioni di una grandezza fisica di riferimento. In entrambi i casi siamo interessati allo sviluppo temporale dell'ampiezza del segnale, che viene valutato a intervalli di tempo regolare (10 ms nell'esempio), solo che nel secondo caso il segnale varia con continuità tra i successivi valori oggetto della misura. Nella parte bassa della figura 2.31 si illustra la rappresentazione delle stesse grandezze numeriche (i valori $0, 5, 10$ e $3, 5, 2$ V) espressa però in binario mediante una notazione posizionale.
I segnali analogici con i quali si ha a che fare sono solitamente del secondo tipo, poiché rappresentano delle variazioni continue nel tempo di una grandezza fisica che vogliamo riprodurre ed eventualmente elaborare su un supporto digitale. L'idea di base per trasformare un segnale analogico in una sequenza di *bit* è allora quella di ricorrere all'impostazione di figura 2.30b, nella quale ad intervalli di tempo regolari si effettua una *lettura* del valore assunto dalla grandezza, prelevando un *campione* della stessa (fig. 2.31a). La lettura del valore del campione viene



(a) Campionamento di un segnale analogico

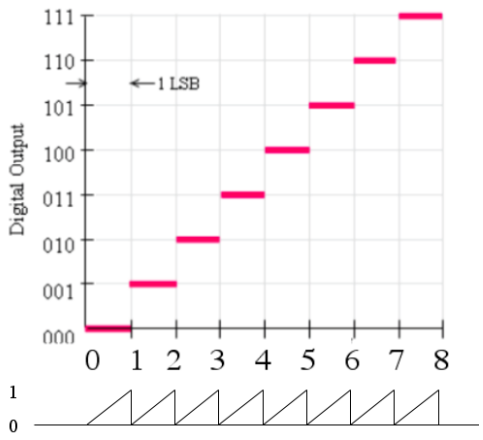
(b) Sequenza dei campioni rappresentati in notazione *floating point*

Figura 2.31: Campionamento di un segnale analogico

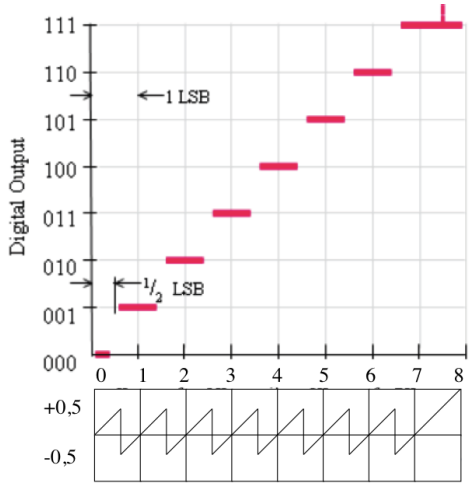
effettuata con uno strumento di misura che sarà caratterizzato da una certa accuratezza, che dipenderà dalla *classe* dello strumento. Se con esso si realizzano delle letture con n cifre (binarie) significative, allora ogni campione potrà essere descritto da un numero in notazione *floating point* con una mantissa di n bit. Ecco allora che il segnale analogico di figura 2.31a viene rappresentato dai campioni di figura 2.31b, che costituiscono una successione binaria in *floating point*. A questo punto è realizzata la *conversione analogico-digitale* (A/D) del segnale.
Il punto critico del ragionamento è che l'operazione di campionamento non consente, in generale, la ricostruzione del segnale analogico originario, poiché ovviamente va persa tutta l'informazione sull'andamento del segnale tra i campioni. Tuttavia, sotto opportune condizioni, tale riconversione diventa possibile. Ciascun segnale analogico è infatti caratterizzato dalla propria *banda*, che rappresenta la massima estensione dello *spettro* di frequenze presenti nel segnale. Se facciamo per esempio riferimento alla voce umana sappiamo che essa si estende tra (circa) 65 e 1500 Hz. Se la *banda* del segnale è limitata, allora si può dimostrare che esiste la possibilità di ricostruire in modo esatto il segnale di partenza a patto di effettuare un campionamento con una frequenza f_c sufficientemente elevata, pari *al doppio* della banda base (*frequenza di Nyquist*). Ciò costituisce il contenuto del celebre *teorema del campionamento di Shannon-Nyquist*, il più importante teorema della teoria dei segnali; esso specifica che, detta B la banda del segnale, la condizione per poter effettuare un campionamento senza perdita d'informazione è che valga la seguente disuguaglianza

$$f_c \geqslant 2 \cdot B$$

Per esempio, nel caso del campionamento di un segnale ad alta fedeltà caratterizzato da una banda che si spinge fino a 22 kHz, la frequenza di campionamento è di $44,1$ kHz; ciò rappresenta uno standard per i CD audio musicali. Il numero di *bit* necessari per effettuare la codifica è legato alla precisione nella lettura dei campioni. Se usiamo a scopo illustrativo i tre *bit* della figura 2.32a, che codificano tutti gli interi tra 0 e 7, possiamo notare che i valori



(a) Conversione analogico digitale e relativo errore di quantizzazione



(b) Conversione analogico digitale a *mezzo bit* e relativo errore di quantizzazione

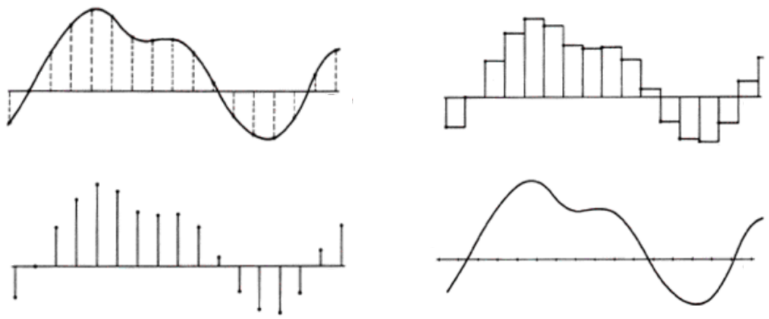
Figura 2.32: Conversione analogico digitale a *bit* pieno e a mezzo *bit*

della lettura compresi tra x e $x+1$ vengono codificati con l'intero $\lfloor x \rfloor$. Ciò determina un *errore di quantizzazione*, dato dalla differenza tra $x - \lfloor x \rfloor$, che oscilla tra 0 e 1; il relativo diagramma a dente di sega è rappresentato sotto la figura. Per limitare l'errore di quantizzazione si può ricorrere all'espediente di bilanciare la codifica, centrandola intorno al valore x (codifica a $1/2$ *bit*). In questo modo tutti i valori compresi tra $x - 1/2$ e $x + 1/2$ saranno codificati come x, e ciò limita l'errore di quantizzazione a $\pm 1/2$ (si veda la figura 2.32b).
Poiché l'errore di quantizzazione modifica di fatto il valore del segnale codificato, la successiva riconversione porta alla sovrapposizione tra il segnale iniziale e un *rumore di quantizzazione*. Per diminuirlo il più possibile bisogna aumentare il numero di livelli della discretizzazione, incrementando il numero di bit associati a ogni campione. La tabella di figura 2.33 mostra il valore percentuale dell'errore in funzione del numero di *bit* usati per

Numero di *bit*	Numero di valori	Valore massimo dell'errore di quantizzazione %
1	2	25
2	4	12,5
4	16	3,125
8	256	0,195
12	4096	$12,20 \cdot 10^{-3}$
16	65536	$7,63 \cdot 10^{-4}$

Figura 2.33: Valore percentuale dell'errore di quantizzazione in funzione del numero di bit e dei livelli usati

la conversione. Dal punto di vista operativo i campioni non rimangono segnali impulsivi isolati (fig.2.34a), ma il loro valore viene mantenuto fino al campione successivo, così come appare in figura 2.34b. L'applicazione di un successivo *filtro passa basso*, che tagliando le alte frequenze tende a smussare le variazioni brusche, restituisce il segnale di partenza, completando la cosiddetta ri-*conversione digitale-analogica* (D/A) del segnale. Quando si ha a disposizione la versione codificata in binario del segnale analogico, è possibile fare su di essa tutte le elaborazioni



(a) Campioni ideali ricavati dal segnale analogico

(b) Rappresentazione del segnale effettivo e successiva riconversione D/A mediante impiego di un filtro passa basso

Figura 2.34: Fasi della conversione A/D e della successiva riconversione D/A

che fossero eventualmente necessarie. La figura 2.35 ci mostra il processo completo di conversione e riconversione di un segnale analogico.

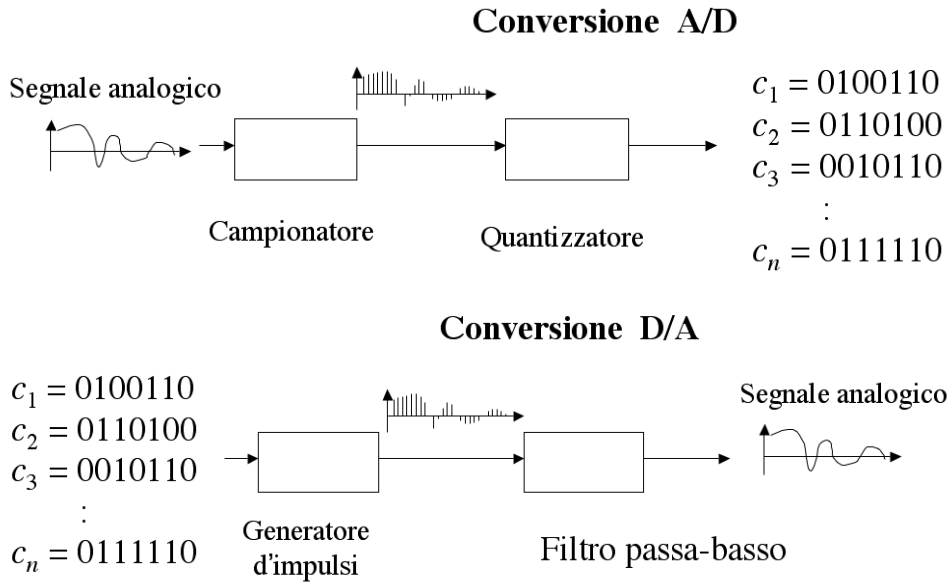


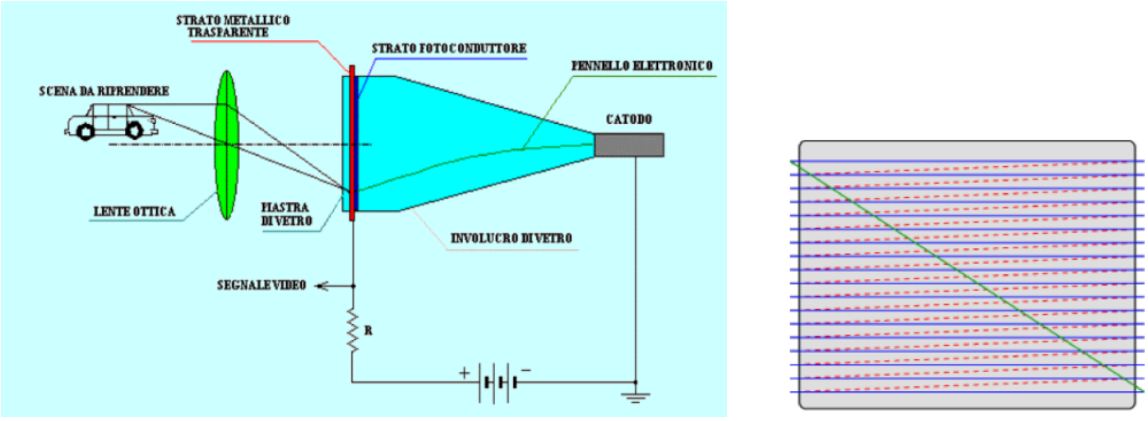
Figura 2.35: Conversione e riconversione di un segnale analogico

2.3.4 La codifica delle immagini

Le immagini sono l'ultima tipologia di informazione che dobbiamo trattare. Prima di addentrarci nel problema della loro trasformazione in *bit*, vale forse la pena raccontare l'evoluzione tecnologica nella loro rappresentazione. Come per altre classi di segnali (telefonici, video, radio,...) si è assistito a un trapasso tra tecnologie analogiche e tecnologie digitali, che in questo caso ha però cambiato anche la struttura fisica e la logica di funzionamento degli schermi usati per la rappresentazione delle immagini.

Elaborazione analogica delle immagini

Un'immagine, così come percepita dal nostri organi di senso, è un "segnale" tipicamente analogico; anche lasciando perdere il problema della rappresentazione tridimensionale degli oggetti, che il nostro cervello riesce a organizzare sfruttando la visione binoculare, e limitandoci a una visione piatta dell'immagine, si ha a che fare con un segnale bidimensionale, che può variare in intensità e cromatismo in modo continuo e lungo tutte le direzioni. Questo aspetto venne sfruttato fin dall'inizio nelle prime telecamere, che usavano un *tubo a raggi catodici* (o CRT, acrostico di *Cathode Ray Tube*) per fare una scansione continua dell'immagine da riprodurre. In questa telecamera un fascio o *pennello elettronico* generato da un catodo (chiamato anche *cannone elettronico*) viene attirato da uno schermo ad alto potenziale (anodo). Sulla parte interna dello schermo, realizzato in vetro, è depositato uno strato sottile di elementi e composti chimici fotoconduttivi (p.es. il *selenio*). L'immagine che si staglia sullo schermo, modulata in intensità luminosa, induce una modulazione analoga anche sullo strato fotoconduttivo, e il pennello elettronico che colpisce lo strato genera una corrente che dipende dallo stato di conduzione (in pratica dalla resistenza) del recettore fotoconduttivo (fig.2.36a). Se al pennello viene impresso una scansione sistematica lungo

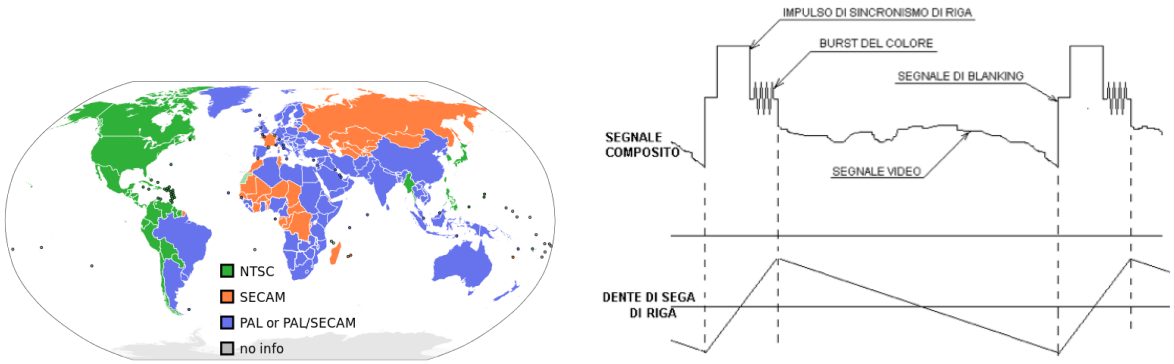


(a) Principio di funzionamento di una telecamera analogica

(b) Scansione di un quadro completo da parte del pennello elettronico

Figura 2.36: Acquisizione di un'immagine con una telecamera analogica

n linee orizzontali appaiate (righe blu della figura 2.36b), si compie una scansione completa dell'immagine (o *quadro*). Quando il pennello elettronico ha esaurito una riga e deve essere riposizionato all'inizio della successiva, allora è necessario spegnerlo (righe rosse tratteggiate). Alla fine del primo quadro il pennello si trova in basso a destra dello schermo, ed è necessario riportarlo nella posizione in alto a sinistra. Nel fare questo bisogna oscurare nuovamente il pennello (riga verde).
Per poter avere un'immagine stabile si deve provvedere a costruire molti quadri al secondo, tipicamente 50 o 60, a seconda dello standard. Il numero di linee da usare per ogni quadro, il numero di quadri per ogni secondo e le modalità di codifica dei colori sono oggetto di tre standard specifici, adottati inizialmente per le trasmissioni televisive; essi sono: il sistema *PAL*, sviluppato in Germania dalla Telefunken e usato principalmente in Europa (a parte la Francia) e in gran parte del mondo; il sistema *SECAM*, sviluppato in Francia e usato anche nella ex Unione Sovietica e in alcune nazioni dell'Africa; il sistema *NTSC*, sviluppato negli USA e impiegato anche in Giappone (fig.2.37a). Mentre i due sistemi europei *PAL* e *SECAM* adottano 625 righe e una frequenza di quadro di 50Hz, lo standard americano *NTSC* usa 525 righe e 60 Hz.
Il segnale di ogni riga, modulato in intensità luninosa, è dunque un segnale analogico simile a quello rappresentato in figura 2.37b, nel quale sono evidenziate anche le informazioni necessarie per attuare il sincronismo tra le righe. Se invece pensiamo allo sviluppo verticale dell'immagine, appare che la stessa viene discretizzata dalle righe, poiché esse sono in numero finito.
Il segnale video generato dalla telecamera modula in ampiezza una portante (onda elettromagnetica ad elevata frequenza) e viene trasmesso mediante antenne agli apparati riceventi. Una volta ricevuta la portante, a seguito della sua demodulazione si riottiene il segnale video originale, che va a comandare un dispositivo di visualizzazione

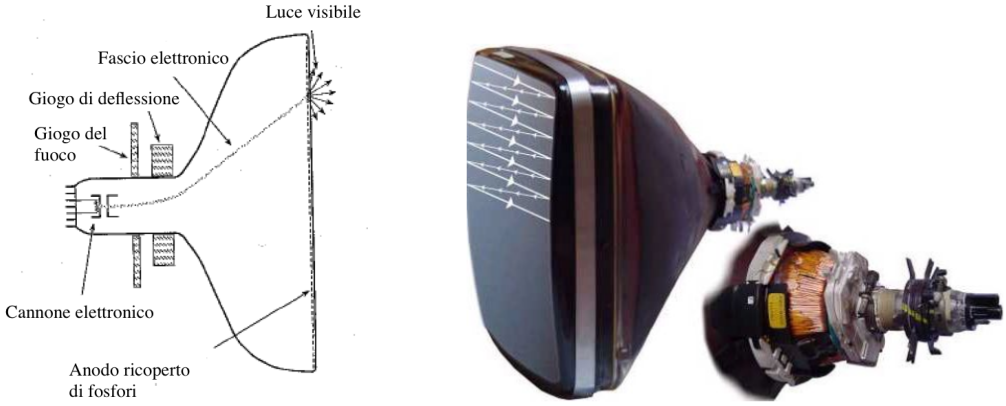


(a) Diffusione degli standard televisivi PAL, SECAM e NTSC

(b) Segnale video

Figura 2.37: Segnale video e relativi standard televisivi

che si basa ancora sulla tecnologia CRT. La ricostruzione dell'immagine avviene con la stessa tecnica usata per la sua scansione, cioè quella delle righe interlacciate che formano un quadro e con la ripetizione dei quadri secondo lo standard dei 50 o 60 Hz. Anche in questo caso c'è un pennello elettronico e un anodo (fig.2.38a); il pennello descrive la scansione delle righe interlacciate e dei quadri usando la stessa matrice della figura 2.36b. La scansione

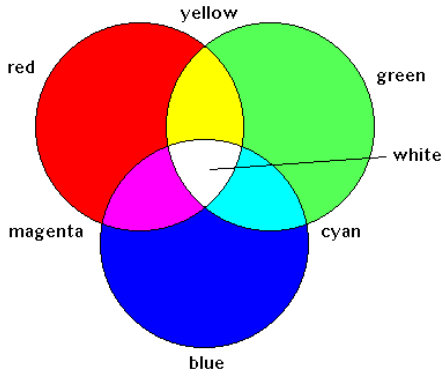


(a) Principio di funzionamento di un tubo CRT

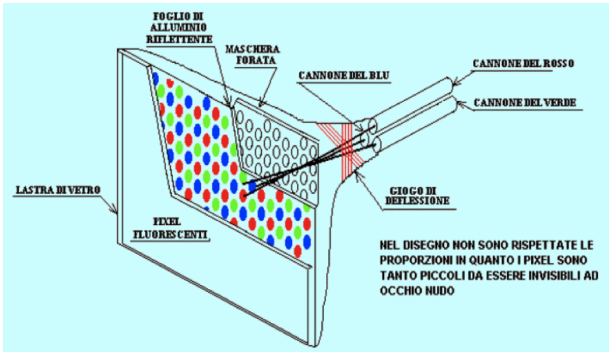
(b) Giogo di deflessione e di fuoco di un tubo CRT

Figura 2.38: Tubo a raggi catodici (CRT) e relativi gioghi di deflessione

della riga da parte del pennello e la successiva concatenazione tra le righe viene attuata da delle bobine (*gioghi di deflessione*, fig.2.38b) che realizzano un campo magnetico in grado di deflettere in modo opportuno il fascio elettronico prima che questa raggiunga lo schermo; l'unica differenza rispetto al tubo della telecamera è che ora il pennello deve generare una luminosità proporzionale a quella dell'immagine. Ciò si ottiene colpendo col pennello un sottile strato di fosfori depositati sulla parte interna dello schermo. L'energia del fascio di elettroni del pennello fa compiere un salto quantico agli elettroni delle orbite più esterne degli atomi dei fosfori, i quali decadendo a livelli quantici di minore energia generano una radiazione luminosa che viene percepita oltre il vetro del tubo CRT (fig.2.38a). Si noti che in ogni istante, a rigore, è acceso un solo punto dell'immagine, ma la visibilità complessiva della stessa e la sua apparente stabilità è dovuta solamente a un effetto ottico legato alla persistenza della retina e a quella dei fosfori che sono usati per la rappresentazione dell'immagine. Tutto quanto detto finora va bene per una trasmissione a livelli di grigio (o televisione bianco e nero), nella quale il segnale video descrive solo la luminosità dell'immagine, ma non distingue i colori. La creazione dei colori avviene con l'espediente di intro-



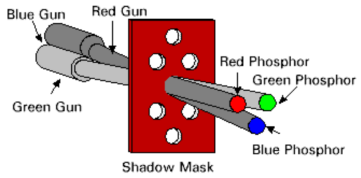
(a) La combinazione dei tre colori base, rosso, verde e blu, in proporzioni opportune consente di creare qualunque colore



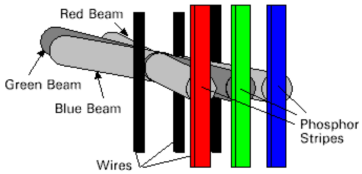
(b) Il tubo CRT a colori è caratterizzato dall'impiego di tre segnali video, uno per ciascun colore

Figura 2.39: Creazione dei colori e principio di funzionamento del tubo CRT a colori

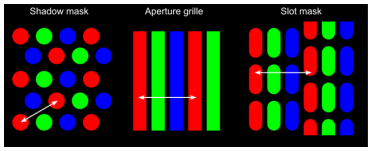
durre tre segnali video contemporanei, associati ai colori base RGB (rosso, verde, blu) che si possono combinare in proporzioni opportune per generare un qualunque colore dello spettro del visibile (fig.2.39a). Di conseguenza il tubo CRT deve avere tre cannoni e tre pennelli elettronici, uno per ciascun colore (fig.2.39b). Per poter creare l'effetto colore che deriva dal mescolamento delle tre componenti, si sfrutta la tendenza dell'occhio a mescolare i colori base quando questi sono percepiti a una distanza opportuna. In altre parole i colori generati dai fosfori dello schermo sono sempre e solo i tre colori base, che vengono percepiti come un unico colore combinazione dei tre se le dimensioni dei punti luminosi sono trascurabili rispetto alla distanza alla quale si osserva l'immagine sullo schermo. I fosfori sullo schermo vengono allora depositati su una griglia che può assumere diverse forme a seconda dello standard usato, come si vede in figura 2.40. Lo standard classico è quello denominato *Shadow mask*



(a) Maschera standard per la deposizione dei fosfori (*Shadow mask*)



(b) Maschera dello standard *Trinitron* denominata *Aperture grille*



(c) Differenza nella visualizzazione dei tre standard

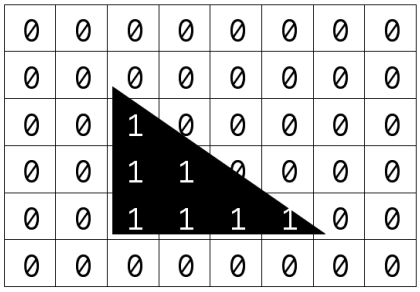
Figura 2.40: Standard in uso per la costruzione del colore a partire dalla posizione dei fosfori

(fig.2.40a), introdotto dalla *RCA* nel 1953 e basato su una maschera forata in cui i tre puntini luminosi RGB sono disposti sui vertici di un triangolo. Lo svantaggio principale di tale soluzione è che circa il 25% dell'energia del fascio va dispersa sulla maschera. Il secondo standard, realizzato dalla Sony nel 1966, si basa su delle linee RGB verticali affiancate (fig.2.40b), perfezionate successivamente dallo standard *slot mask* introdotto successivamente dalla *NEC* (fig.2.40c).
La tecnica della manipolazione analogica delle immagini e la tecnologia degli schermi CRT sono oramai superate.

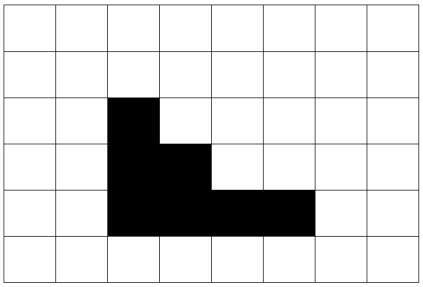
Elaborazione digitale delle immagini

Un approccio completamente diverso alla rappresentazione e alla elaborazione delle immagini emerge a partire dai primi anni '70, grazie allo sviluppo dei microprocessori. Poiché si deve pervenire a una digitalizzazione dell'immagine, l'idea è quella di suddividere la stessa in una griglia di n colonne e m righe, in modo da rappre-

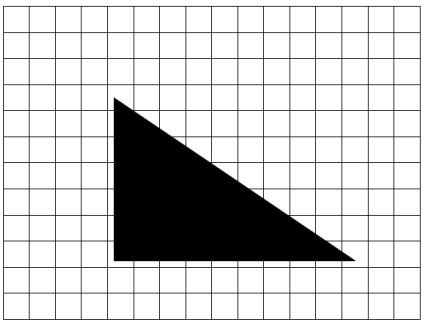
sentarla mediante $n \cdot m$ elementi d'immagine chiamati *pixel* (che deriva da *picture elements*). Una volta costruita la griglia si può attribuire, nel caso di un'immagine a bianco e nero, il valore 1 a un pixel che copre una parte nera dell'immagine e 0 a un pixel che copre una parte bianca. Nel caso in cui un pixel copra contemporaneamente una parte bianca e una nera si può decidere sulla base della maggioranza della superficie. In figura 2.41a si vede un'immagine di 8 colonne e 6 righe, cioè con 48 pixel di *risoluzione*, che rappresenta un semplice triangolo in bianco e nero. Si noti che una volta attribuito il valore 0 o 1 al *pixel* sulla base della predominanza del bianco o del nero, la decodifica porta a una figura (fig. 2.41b) che risulta frastagliata a causa della bassa risoluzione dell'immagine. Aumentando la finezza della grana, e quindi il numero di righe e di colonne (che ora sono $16 \cdot 12$), la qualità



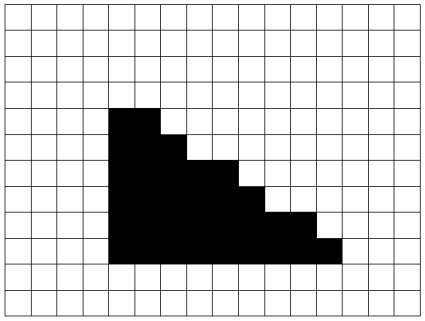
(a) Codifica di un'immagine a bianco e nero con bassa risoluzione



(b) La relativa decodifica genera un'immagine frastagliata



(c) Codifica della stessa immagine con una risoluzione più elevata

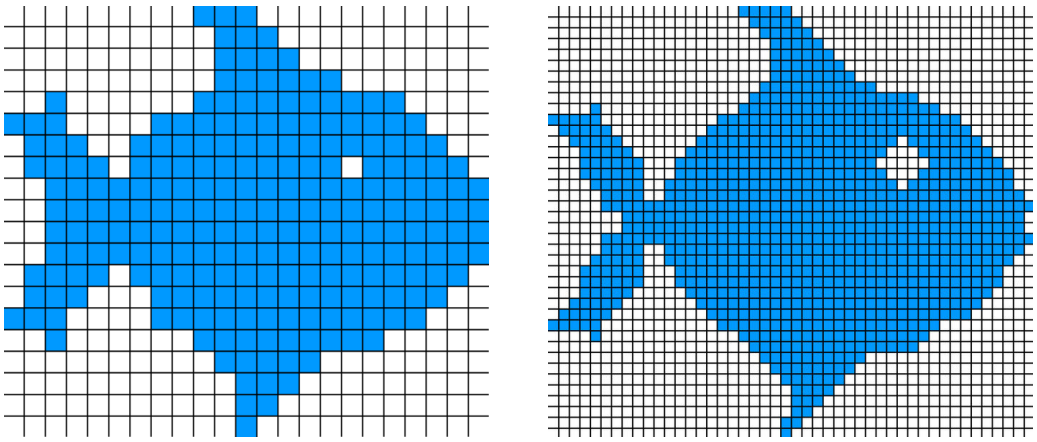


(d) Aumentando la risoluzione la qualità dell'immagine migliora

Figura 2.41: Gestione digitale delle immagini mediante matrice di *pixel*

dell'immagine migliora di conseguenza (fig. 2.41c e 2.41d). La contropartita nel miglioramento della qualità è data dall'aumento considerevole della quantità di memoria associata all'immagine.
La figura 2.42 mostra la variazione della qualità dell'immagine su un esempio un po' più articolato, basato su una risoluzione $23 \cdot 20 = 460$ *pixel* per la figura 2.42a e $46 \cdot 40 = 1840$ *pixel* per la figura 2.42b. Se ipotizziamo una figura quadrata di dimensione $n \cdot n$, il numero totale di *pixel* cresce dunque con il *quadrato* della dimensione del lato.
Le immagini viste finora sono caratterizzate da due soli livelli di colore, bianco e nero; pur restando nell'ambito delle immagini *monocromatiche* possiamo migliorare la loro granularità e la percezione delle sfumature ammettendo una scala di grigi che vada dal bianco brillante al nero. Per distinguere tra due livelli (bianco e nero) basta un singolo *bit*; se vogliamo invece distinguere tra M livelli servono $\lceil \log_2 M \rceil$ *bit*. Se poniamo $M = 2^b$ serviranno b bit. La figura 2.43 ci mostra un esempio con $b = 4$ e $2^4 = 16$ livelli di grigio, assieme alle rispettive codifiche espresse in numero decimale, esadecimale (cioè in base 16) e binario. L'introduzione di 2^b livelli di grigio aumenta l'occupazione di memoria dell'immagine, com'era lecito aspettarsi, visto che ora a ogni *pixel* sono associati b *bit* di memoria; i *bit* totali associati all'immagine sono allora $m \cdot n \cdot b$ e con essi è possibile realizzare $2^{m \cdot n \cdot b}$ possibili immagini diverse. I valori più comuni di b usati in passato per le scale di grigio erano 8, che corrispondono a 1 byte e che consentono 256 livelli di grigio, e 16, che consentono 65536 livelli di grigio.

Dal punto di vista tecnico possiamo fare le seguenti due osservazioni; la prima è che il *pixel* risulta essere la più piccola zona *indirizzabile* dell'immagine, dove per indirizzabile s'intende il fatto che il processore ha su di essa un controllo totale, potendo variare l'intensità di grigio (o il colore) del singolo *pixel* in modo indipedente dai *pixel* adiacenti. La seconda osservazione è che nel caso monocromatico la variazione della tonalità di grigio è legata essenzialmente alla variazione di luminosità del *pixel*. Quando si vuole introdurre il colore nelle immagini digitali bisogna ricorrere alla tecnica della ricostruzione RGB dello stesso basata sui tre colori base, rosso, verde e blu, di cui abbiamo già discusso precedentemente con riferimento ai monitor CRT (vedi fig. 2.39a). In questo caso il *pixel* è suddiviso in tre *subpixel* colorati, uno per ogni componente, ciascuno dei quali viene comandato dalla corrispondente scala di livelli di emissione monocromatica. Poiché i *subpixel* sono molto piccoli, l'effetto è quello di un colore dato dalla mescolanza dei tre colori base quando la distanza da cui si osserva il *pixel* è molto maggiore della sua dimensione. La figura 2.44a ci mostra la struttura dei *subpixel* su un moderno schermo piatto.
Nei vecchi monitor CRT, basati sulle schede grafiche denominate VGA, la *profondità di colore* era di 8 *bit* per ciascun pixel (8 *bit* color), di cui 3 per il rosso e il verde (8 livelli) e 2 per il blu (4 livelli); la scelta di usare solo due bit per il blu è legata al fatto che l'occhio umano ha una minore sensibilità su questa lunghezza d'onda. Nelle codifiche successive si è passati prima a 16 *bit* per pixel (*High color*) e successivamente a $24, 30, 32, 36, 48, 64$ *bit* per pixel. Nel caso del sistema *High color* vengono assegnati 5 *bit* per rosso e blu e 6 *bit* per il verde, cui corrisponde una migliore sensibilità del sistema visivo umano. Il sistema *Truecolor* è invece caratterizzato da 8 *bit* e 256 livelli possibili per ciascun canale RGB. La figura 2.44b ci consente di vedere come vengono creati i colori *Truecolor* calibrando le diverse componenti RGB nell'intervallo $[0 - 255]$. Le altre combinazioni possibili sono riportate nella tabella di figura 2.45. La presenza di un certo numero di *bit* per il cosiddetto *Canale* α consente di mescolare i colori dell'immagine da riprodurre con quelli dello sfondo, in modo da dare l'effetto di una *trasparenza* più o meno marcata. La figura 2.46 ci fa vedere la scomposizione di un'immagine nelle componenti RGB, mentre la 2.47 ci mostra come migliori la qualità dell'immagine all'aumentare dei livelli di colore.
Il progressivo aumento della risoluzione e della dimensione degli schermi ha portato a un significativo miglioramento della qualità delle immagini, che sono passate da una risoluzione 640×200 delle prime schede video CGA, introdotte da IBM nel 1981, ai 3840×2400 *pixel* dello standard WQUXGA dei più grandi monitor ad oggi mai costruiti. La tabella di figura 2.48 ci mostra gli standard principali che si sono succeduti nel corso degli anni. Si osservi che la seconda parte della tabella, da HXGA in poi, fa riferimento a standard che non sono stati ancora impiegati in monitor commerciali, ma che sono in già in uso su sensori di alcune fotocamere. L'ultima riga mette in evidenza le caratteristiche del sensore commerciale a massima risoluzione oggi esistente, da $8176 \times 6132 = 50,14$ Mega *pixel*, che equipaggia la fotocamera *Hasselblad H4D*, ma che non fa riferimento ad alcun standard.
Questo aumento impressionante nella qualità dell'immagine, oramai molto vicina alla granularità ottenibile dallo sviluppo dell vecchie pellicole, ha tuttavia un costo rilevante in termini di memoria occupata. Nell'ultima colonna

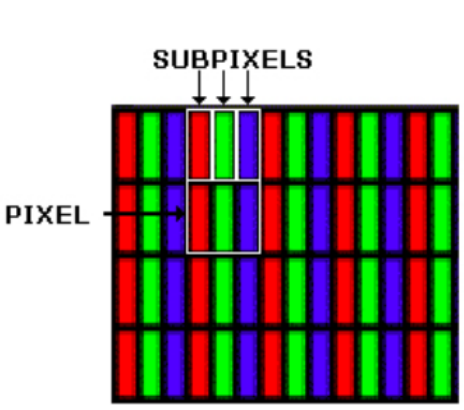


(a) $23 \cdot 20 = 460$ *pixel* (b) $46 \cdot 40 = 1840$ *pixel*

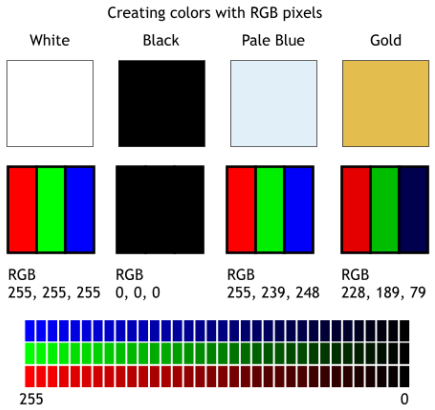
Figura 2.42: Miglioramento della qualità dell'immagine all'aumento della risoluzione

Livello di grigio																
Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Figura 2.43: Scala dei grigi ricavata da 4 *bit* e 2^4 livelli



(a) Struttura dei *subpixel* che realizzano l'effetto cromatico del colore mediante la mescolanza delle tre componenti RGB



(b) Composizione dei colori con lo schema *Truecolor*; per ogni canale RGB si hanno a disposizione i livelli dell'intervallo $[0 - 255]$

Figura 2.44: Costruzione dei colori mediante *subpixel* RGB

Bit per *pixel*	R	G	B	Canale α	Numero di livelli	Nome
8	3	3	2	-	256	8 *bit* color
16	5	6	5	-	65536	High color
24	8	8	8	-	16777216	True color
32	8	8	8	8	16777216	True color
30	10	10	10	-	$1,073 \cdot 10^9$	Deep color
36	12	12	12	-	$68,7 \cdot 10^9$	Deep color
48	16	16	16	-	$281,5 \cdot 10^{12}$	Deep color
64	16	16	16	16	$281,5 \cdot 10^{12}$	Deep color

Figura 2.45: Principali codifiche RGB

della tabella di figura 2.48 troviamo il peso in *MegaByte* delle singole immagini associate ai vari standard. Possiamo allora notare che un'immagine XGA ha 768 kB, che corrisponde a poco più della metà della capacità dei vecchi *floppy-disk* da $1,44$ MB. Per un'immagine 3840×2400 in formato WQUXGA servono oltre 35 MB, mentre l'immagine fissata dalla *Hasselblad H4D* necessita di oltre 191 MB. È vero che molto spesso si può ricorrere a una compressione del file corrispondente, che però funziona bene solo con immagini piuttosto uniformi. Se l'immagine è molto complessa e con sfondi irregolari la compressione porta a risultati modesti, a meno che non venga effettuata una compressione *psicovisuale*, che comporta però una certa perdita d'informazione (l'immagine decompressa non è più identica all'originale). Se inoltre siamo di fronte a immagini nell'ambito medico non è neanche lecito effettuare una compressione di questo genere, poichè una perdita d'informazione visuale sarebbe eticamente inaccettabile.

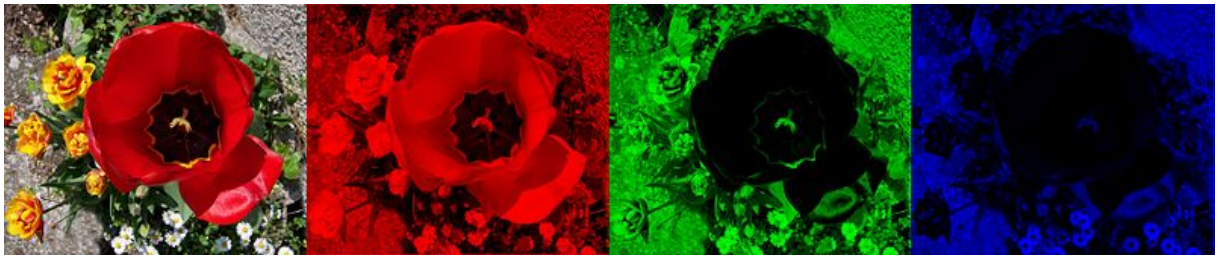
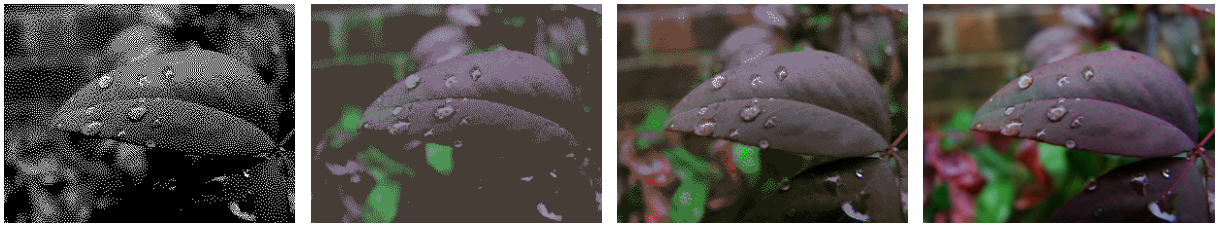


Figura 2.46: Scomposizione di un'immagine nelle componenti RGB



(a) Immagine monocromatica con 2 livelli di colore, bianco e nero (1-*bit* image)

(b) Immagine con 4 livelli di colore (2-*bit* image)

(c) Immagine con 16 livelli di colore (4-*bit* image)

(d) Immagine con 256 livelli di colore (8-*bit* image)

Figura 2.47: Miglioramento della qualità dell'immagine all'aumentare dei livelli di colore

Sigla	Risoluzione	Nome	Aspect Ratio	Bit	M*Pixel*	Memoria
CGA	640×200	Color Graphics Adapter	16:5	1	0,13	15,63 kB
EGA	640×350	Enhanced Graphics Adapter	64:35	4	0,22	109,38 kB
VGA	640×480	Video Graphics Array	4:3	4	0,31	150 kB
SVGA	800×600	Super VGA	4:3	4	0,48	234 kB
XGA	1024×768	Extended GA	4:3	8	0,79	768 kB
WXGA	1280×768	Wide XGA	5:3	32	0,98	3,75 MB
SXGA	1280×1024	Super XGA	5:4	32	1,31	5,00 MB
WSXGA+	1680×1050	Wide SXGA Plus	16:10	32	1,76	6,73 MB
UXGA	1600×1200	Ultra XGA	4:3	32	1,02	7,32 MB
WUXGA	1920×1200	Wide Ultra XGA	16:10	32	2,30	8,79 MB
QXGA	2048×1536	Quad XGA	4:3	32	3,15	12 MB
QSXGA	2560×2048	Quad Super XGA	5:4	32	5,24	20 MB
WQSXGA	3200×2048	Wide QSXGA	25:16	32	6,55	25 MB
WQUXGA	3840×2400	Wide Quad Ultra XGA	16:10	32	9,22	35,16 MB
HXGA	4096×3072	Hexadecatuple XGA	4:3	32	12,58	48 MB
WHXGA	5120×3200	Wide HXGA	16:10	32	16,38	62,5 MB
HSXGA	5120×4096	Hexadecatuple SXGA	5:4	32	20,97	80 MB
WHSXGA	6400×4096	Wide Hexadecatuple SXGA	25:16	32	26,21	100 MB
HUXGA	6400×4800	Hexadecatuple UXGA	4:3	32	30,72	117,19 MB
WHUXGA	7680×4800	Wide Hexadecatuple UXGA	16:10	32	36,86	140,63 MB
Sensore	8176×6132	Fotocamera Hasselblad H4D	4:3	32	50,14	191,25 MB

Figura 2.48: Principali standard di risoluzione per monitor e loro caratteristiche

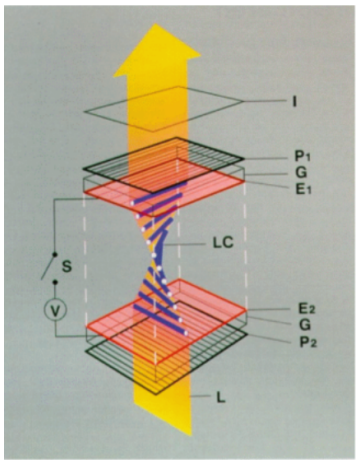
Tipo di immagine	Risoluzione in *Pixel*	*Bit* per *Pixel*
Scintillation camera (planar)	64×64 o 128×128	8 o 16
SPECT	64×64 o 128×128	8 o 16
PET	128×128	16
Digital fluoroscopy, cardiac catheter lab	512×512 o 1024×1024	8 o 12
Computed radiography, digitalized chest film	2000×2500	$10 - 12$
Mammography $(18 \times 24)(24 \times 30)$	da 1800×2300 a 4800×6000	$12 - 16$
X-ray CT	512×512	12
MRI	da 64×64 a 1024×1024	12
Ultrasound	512×512	8
Abbreviazioni: SPECT - *Single Photon Emission Computed Tomography*; CT - *Computed Tomography* PET - *Positron Emission Tomography*; MRI - *Magnetic Resonance Imaging*		

Figura 2.49: Valori tipici di risoluzione e di profondità del colore per immagini radiologiche

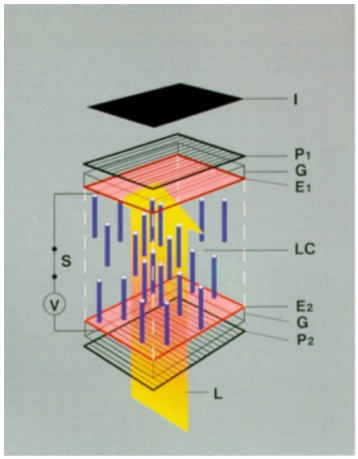
Quando si ha a che fare con le immagini i calcolatori devono essere dotati di adeguate caratteristiche tecniche per la lor gestione efficiente, cioè avere una memoria primaria (RAM) e una memoria di massa (HD) molto corpose, processori (o multiprocessori) e connessioni di rete molto veloci, altrimenti tutto si rallenta in modo inaccettabile. Per quanto riguarda le immagini in ambito radiologico non esistono degli standard specifici, e ogni costruttore propone apparecchiature con caratteristiche specifiche. Esiste tuttavia un ambito di risoluzioni che è tipico per la tecnologia impiegata per acquisire l'immagine, e la tabella di figura 2.49 ci mostra alcuni tra questi valori.

La tecnologia degli schermi digitali

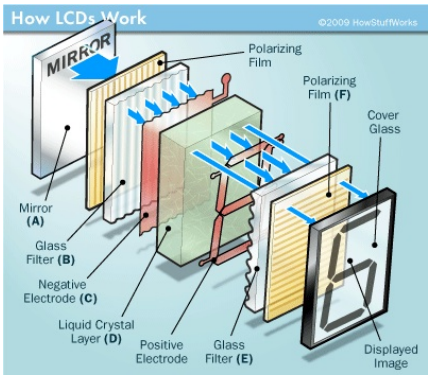
Il trapasso tra tecnologia analogica e digitale nell'ambito dell'elaborazione delle immagini è stato affiancato da una (quasi) contemporanea transizione tra monitor CRT a raggi catodici e schermi piatti basati sulla tecnologia LCD dei *cristalli liquidi* (*Liquid Crystal Display*). La tecnologia LCD sfrutta la proprietà di alcuni composti organici (bifenile) di modificare la propria trasparenza in funzione di un campo elettrico applicato. Più precisamente



(a) Stato OFF di un cristallo LCD *Twisted Nematic* (assenza di campo elettrico)



(b) Stato ON di un cristallo LCD *Twisted Nematic* (campo elettrico applicato)

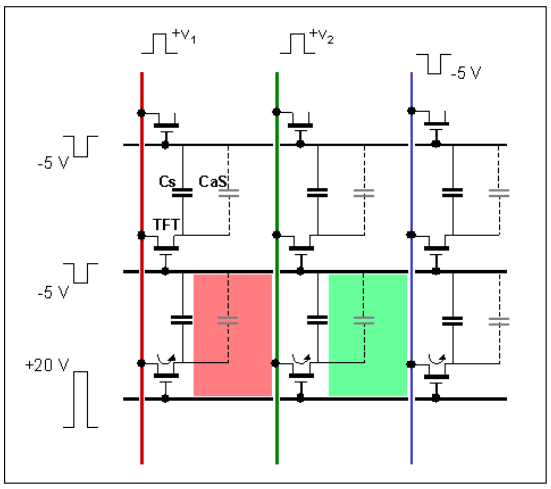


(c) Funzionamento di un cristallo LCD di tipo alfanumerico

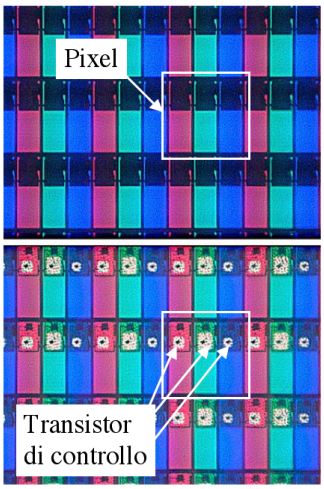
Figura 2.50: Funzionamento di uno schermo LCD a cristalli liquidi in modalità di cristallo normalmente chiaro

si sfrutta il cosiddetto *effetto nematico ritorto* o *Twisted Nematic* (TN), nel quale le molecole del bifenile risultano disposte lungo una configurazione a elica (bastoncini blu di fig.2.50a) e il passaggio della luce viene comandato da un campo elettrico. Il cristallo liquido (LC) è contenuto tra due elettrodi trasparenti (E_1, E_2), due vetri (G) e due schermi polarizzati (P_1, P_2). Quando nel cristallo non c'è un campo elettrico applicato si è nello stato OFF, la luce supera il primo polarizzatore P_2, passa attraverso il cristallo liquido e subisce una progressiva rotazione di $90°$ della polarizzazione, uscendo dallo schermo polarizzatore P_1 ortogonale a P_2. Quando invece viene applicato un campo elettrico (stato ON), le molecole del bifenile si orientano tutte lungo le linee del campo, la luce non subisce alcuna rotazione della polarizzazione e viene poi bloccata dal polarizzatore P_1, a essa ortogonale (fig.2.50b), facendo apparire il cristallo scuro. Questa modalità di funzionamento corrisponde a un cristallo *normalmente chiaro* (cioè in assenza di campo elettrico). Per cambiare logica di funzionamento (cristallo *normalmente scuro*) è sufficiente orientare di $90°$ uno dei due schermi polarizzatori, in modo che siano tra loro paralleli invece che ortogonali.

Gli schermi LCD, introdotti dalla NEC nel 1986, non generano una luce propria, ma sfruttano quella ambien-



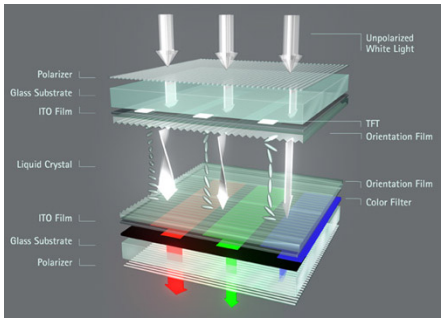
(a) Struttura della matrice TFT che comanda i *subpixel* RGB in un cristallo liquido *Twisted Nematic* normalmente scuro



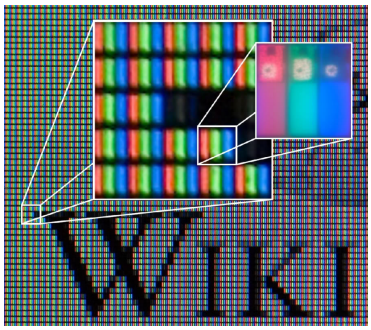
(b) Gruppo di *subpixel* con i relativi transistor di comando

Figura 2.51: Struttura della matrice dei *subpixel*

tale che viene riflessa da una superficie speculare posta dietro al cristallo (il "mirror" di fig.2.50c); per questo motivo vengono detti anche schermi *a matrice passiva*. In tale tipo di schermo ogni elemento viene comandato direttamente dalla tensione che serve per farlo funzionare. Questa soluzione rappresenta però una limitazione alla costruzione di schermi con elevata risoluzione, poiché sarebbero necessarie milioni di connessioni per comandare correttamente tutti i singoli *pixel*. Per questo motivo a partire dagli anni '80 si sviluppò la tecnologia TFT-LCD a *matrice attiva* (Thin-Film-Transistor LCD), nella quale il potenziale elettrico di ciascun *pixel* è comandato da un transistor (fig.2.51a). Tutti i transistor (del tipo MOS-FET) sono collegati per righe, tramite l'elettrodo d'ingresso denominato *gate*, e per colonne, tramite l'elettrodo comune chiamato *source*. Le righe vengono alimentate secondo una sequenza temporale, una dopo l'altra, con un impulso $+20$ visibile in figura 2.51a, in modo da portare nella posizione ON tutti i transistor e consentire alle colonne dei dati di trasmettere le eventuali cariche agli elettrodi C_aS del cristallo liquido. Se la carica è presente (segnali $+V_1$ e $+V_2$) il cristallo TN (normalmente scuro) si polarizza, passa nello stato ON, diventa trasparente e la luce passa attraverso il filtro colorato corrispondente, rosso e verde per i *subpixel* della figura 2.51a, che genereranno quindi il colore giallo. Se la carica non è presente (segnale $-5V$) il cristallo TN rimane scuro. Una volta caricato, il singolo *subpixel* mantiene la carica grazie alla capacità C_s e alla capacità parassita C_aS, almeno fino a quando arriva il nuovo comando di riga. Se il pixel deve nel frattempo variare stato, il nuovo comando di colonna sarà del tipo $-5V$ e porterà all'oscuramento dello stesso. La figura 2.52 mostra un'esploso completo di uno schermo TFT-LCD e l'ingrandimento di un'immagine che evidenzia un gruppo di *pixel* e i transistor dei relativi *subpixel*. Ricordiamo infine che la tecnologia TFT-LCD necessita di una sorgente luminosa applicata sullo sfondo, che può essere basata su una luce fluorescente (*Cold Cathode Fluorescent Light* -

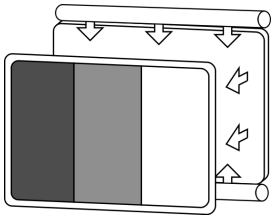


(a) Esploso completo della struttura di uno schermo a matrice attiva del tipo TFT-LCD con cristallo liquido *Twisted Nematic* normalmente scuro

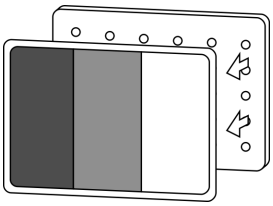


(b) Ingrandimento di un'immagine e relativi *subpixel*

Figura 2.52: Struttura di uno schermo TFT-LCD con particolore dei *subpixel*



(a) Luce di sfondo generata da tubi fluorescenti (CCFL)

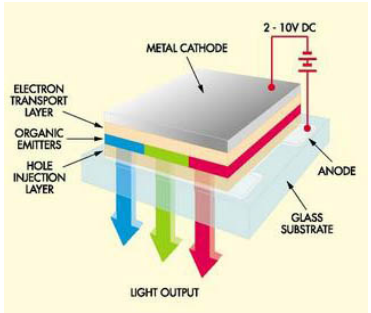


(b) Luce di sfondo generata da LED

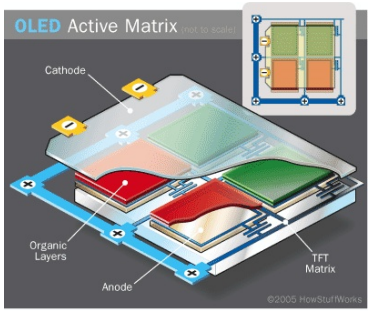
Figura 2.53: Due diverse tecniche per l'illuminazione della matrice TFT-LCD

CCFL) oppure su una luce generata da dei LED (*Light Emitting Diode*) (fig. 2.53).

Recentemente, a partire dal 2000, si è consolidata una nuova tecnologia a matrice attiva denominata OLED (*Organic Light Emitting Diode*); il principio fisico è legato al fenomeno della *elettroluminescenza* di alcuni composti organici e inorganici, vale a dire la loro capacità di emettere una radiazione luminosa quando sottoposti a un campo elettrico o percorsi da una corrente (fig.2.54). In questo modo si evita l'uso della sorgente di luce dello sfondo e il singolo *subpixel* si comporta di fatto come un diodo LED che genera una radiazione rossa, verde o blu, a seconda del composto usato.



(a) Principio di funzionamento di uno schermo OLED



(b) Struttura della matrice di uno schermo OLED

Figura 2.54: Struttura di uno schermo a matrice OLED (*Organic Light Emitting Diode*)

Capitolo 3

Introduzione alle tecnologie elettroniche

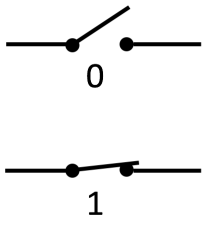
Abbiamo quotidianamente a che fare con una gran quantità di dispositivi elettronici di tutti i tipi; un elenco molto riduttivo comprende

- computer
- *smartphone*
- telefoni fissi
- televisori
- elettrodomestici
- registratori
- radioricevitori
- trasmettitori
- lettori CD, DVD
- telecomandi
- strumenti di misura
- dispositivi di controllo di vario genere (temperatura, pressione, ...)
- dispositivi di sicurezza per automobili (ABS, ASR, ESP, ...)
- dispositivi di allarme
- lettori ottici
- lettori di impronte digitali
- ...

Ormai ci siamo abituati a scorrere le dita per cambiare le pagine su uno *smartphone*, a far riconoscere le nostre impronte digitali per usare i nostri privilegi di accesso, a premere energicamente il pedale del freno della nostra autovettura (quando serve), senza temere che le ruote si blocchino. Tutte queste attività, che oramai diamo per scontate, ci sono diventate familiari grazie a uno straordinario sviluppo della tecnologia elettronica, che non ha paragoni con nessun altro tipo di tecnologia. Nella parte introduttiva, riguardante la storia dell'informatica, e nel capitolo 2 dedicato all'informazione, abbiamo brevemente reso conto di questa rivoluzione microelettronica, che sta incidendo in modo profondo persino nella struttura della nostra società. In questo capitolo vorremmo andare un po' più nello specifico, soprattutto per comprendere le radici concettuali di questo sviluppo tecnologico; queste sono legate in modo essenziale con la logica Booleana da una parte, e con la possibilità di miniaturizzare a livello microscopico i dispositivi che la manipolano dall'altra.

3.1 Il bit (elettro)meccanico: l'interruttore

Il punto di partenza concettuale di tutta la parabola tecnologica che ci ha portato dai primi circuiti elettrici ai moderni e sofisticati dispositivi elettronici è l'*interruttore*. Tutti sanno che questo è il più semplice dispositivo elettrico che si possa concepire, idoneo solamente a interrompere il flusso di una corrente; in figura 3.1 vediamo il suo simbolo elettrico e una realizzazione commerciale. Il motivo per cui l'interruttore è così importante è legato al fatto che esso può assumere due *stati*, e cioè *spento* e *acceso*. Spento significa che la lamina metallica che lo costituisce è alzata e c'è un'interruzione del circuito; acceso significa invece che la lamina metallica è abbassata, il circuito è chiuso ed è possibile un flusso di corrente.

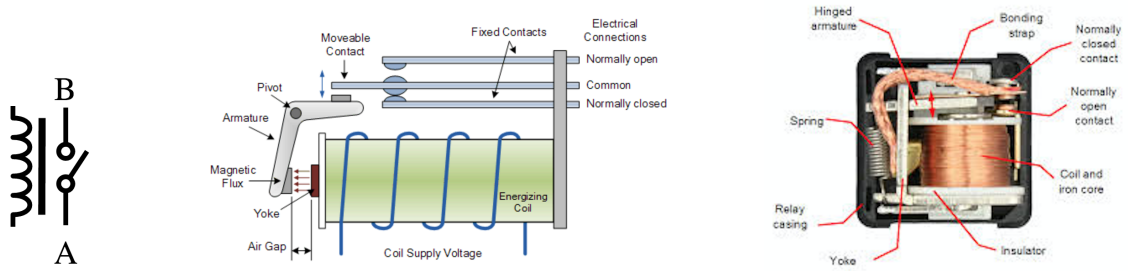


(a) Simbolo elettrico dell'interruttore

(b) Interruttore commerciale

Figura 3.1: Interruttore

E' convenzione usare il simbolo 0 per caratterizzare l'interruttore spento e il simbolo 1 per caratterizzare l'interruttore acceso. Il fatto che il suo funzionamento sia rappresentato dai due stati 0 e 1, consentì a Shannon, come ricordato nel paragrafo 1.1, di agganciarlo alla *Logica Booleana* (o *Logica Binaria*), che è lo strumento concettuale che Boole elaborò nella metà del XIX secolo; vedremo nel seguito come si realizzò questa connessione. Per intanto possiamo anticipare il fatto che i due stati 1 e 0 sono sempre associati alla presenza o all'assenza di una certa grandezza fisica, tipicamente una tensione, una corrente o una carica elettrica, nel nostro ambito elettronico; nel nostro caso 1 significa il fatto che c'è tensione a valle dell'interruttore.
L'interruttore è un dispositivo ad azionamento manuale; questo esclude di poterlo usare in tutte quelle situazioni in cui lo stesso sia inaccessibile, o sia molto distante, oppure si pretenda un funzionamento *automatico* (cioè non manuale). Ecco allora che all'inizio dell'ottocento si costruirono i primi interruttori comandati a distanza, i cosiddetti *relè*, che consentono un controllo *remoto* dell'interruttore mediante l'uso di una corrente elettrica di azionamento. Il principio di funzionamento è il seguente: una corrente (continua o alternata, a seconda dei casi) scorre in una induttanza (detta anche *bobina* o *solenoide*), costituita da un filo di rame avvolto a spire molto strette su un rocchetto di supporto; il valore dell'induttanza viene aumentato in modo considerevole inserendo un nucleo ferromagnetico all'interno della bobina; in tal modo si costruisce un *elettromagnete*.



(a) Simbolo elettrico del relè

(b) Principio di funzionamento di un relè

(c) Relè commerciale

Figura 3.2: Struttura e funzionamento di un relè

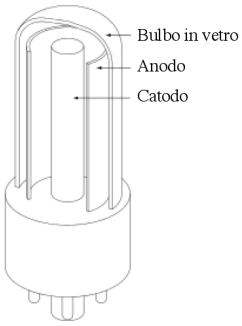
Quando c'è corrente si forma un flusso magnetico intenso all'interno del nucleo ferromagnetico, in grado di attirare un'ancoretta metallica che a sua volta, vincendo la resistenza di una molla, muove l'interruttore. Il relè è dunque un interruttore comandato da una corrente. Poiché i contatti dell'interruttore possono essere anche molto grossi, un relè può essere usato per controllare un grosso flusso di corrente a partire da una piccola corrente, cioè quella necessaria per eccitare l'elettromagnete. Questa è dunque un'altra funzione del relè: con un piccolo interruttore, in grado di sopportare la sola corrente di eccitazione della bobina, siamo in grado di controllare un carico che assorbe una corrente molto grande.
Poiché il relè è un interruttore, sia pure comandato in corrente, anche per esso si possono usare le due costanti binarie 1 e 0 per descrivere il suo funzionamento. Come vedremo nel seguito l'alfabeto binario è il cardine logico di tutta la tecnologia digitale, e infatti i primi calcolatori furono realizzati con relè, anche se questa tecnologia durò molto poco, piché venne quasi subito soppiantata da quella dei tubi termoionici. Il principale svantaggio di questa tecnologia è relativo ai tempi necessari e all'energia necessari per cambiare stato; i tempi sono molti alti, trattandosi di un dispositivo elettromeccanico; pure l'energia è considerevole, poiché bisogna spostare un'ancoretta metallica, vincendo per altro la resistenza di una molla.

3.2 Il bit termoionico: i tubi a vuoto

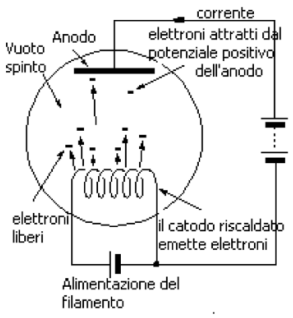
Il secondo passo nello sviluppo tecnologico che portò alla costruzione dei moderni calcolatori si compì con l'invenzione del cosiddetto *tubo termoionico*. Esso è una derivazione diretta della lampadina, poiché è costituito da un bulbo di vetro nel quale viene praticato il vuoto. Al suo interno c'è un filamento metallico, che si riscalda per effetto Joule al passaggio di una corrente elettrica, emettendo una debole luce rossastra. In prossimità del filamento permane una nube elettronica costituita dagli elettroni che sfuggono agli orbitali dei rispettivi atomi; ciò a causa dell'elevata energia cinetica accumulata degli elettroni e dovuta all'agitazione termica derivante dall'alta temperatura del filamento, che nei tubi termoionici essa si attesta intorno ai 450-600 °C. In realtà, se escludiamo i primi modelli di tubo denominati *a riscaldamento diretto*, solitamente il filamento riscalda l'interno di una superficie metallica cilindrica chiamata *catodo* (3.3a), ricoperta di ossidi speciali che facilitano la fuoriuscita degli elettroni dagli orbitali (tipicamente ossidi di cesio).

3.2.1 Il diodo a vuoto

Se ora, all'interno del tubo, mettiamo una placchetta metallica chiamata *anodo*, e la polarizziamo positivamente rispetto al catodo, gli elettroni della nube elettronica vengono attratti dall'anodo e si stabilisce un flusso unidirezionale di corrente (fig. 3.3b).



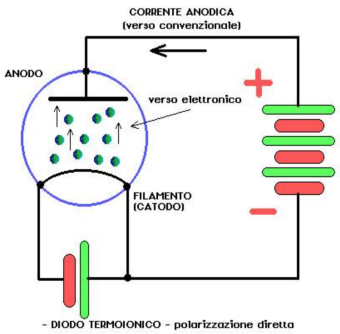
(a) Struttura interna di un diodo a vuoto



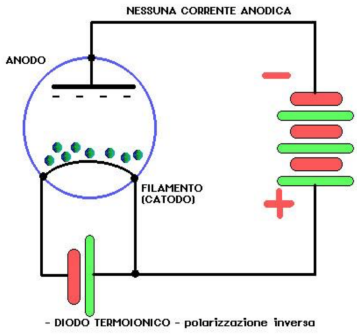
(b) Principio di funzionamento del diodo

Figura 3.3: Struttura e principio di funzionamento del diodo a vuoto

Un dispositivo di questo genere, interessato da un flusso unidirezionale di corrente, si chiama *diodo*; trattandosi di un diodo realizzato con la tecnologia dei tubi termoionici si specifica ulteriormente usando la locuzione *diodo a vuoto*.
Si osservi il diverso comportamento del diodo con l'anodo polarizzato positivamente (figura 3.4a) e negativamente (figura 3.4b). Nel caso di anodo positivo si stabilisce un flusso continuo di elettroni, dal catodo all'anodo, cui corrisponde un flusso convenzionale di corrente nel senso opposto (la convenzione è che il senso di scorrimento è quello delle cariche positive). Se viceversa si polarizza l'anodo negativamente, il flusso di corrente si blocca, poiché il potenziale dell'anodo respinge gli elettroni,



(a) Il diodo conduce, poiché l'anodo è correttamente polarizzato con un potenziale positivo



(b) Il diodo non conduce, poiché l'anodo è polarizzato con un potenziale negativo

Figura 3.4: Il diodo a vuoto conduce solo se correttamente polarizzato

Il legame che esiste tra la tensione e la corrente anodica non è lineare, ma segue la seguente legge di Child-Langmuir

$$I_a = kV_a^{3/2} \tag{3.1}$$

dove la costante k è la *perveanza* del tubo; in figura 3.5 si vede l'andamento della curva per tre diversi diodi a vuoto commerciali.

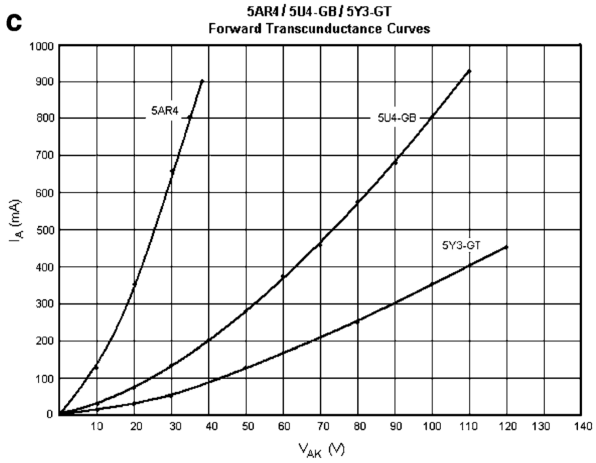
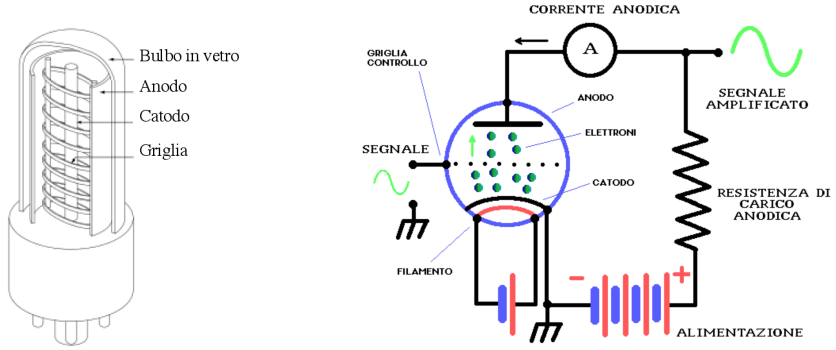


Figura 3.5: Curve caratteristiche di alcuni diodi a vuoto commerciali

I diodi hanno diverse applicazioni, la principale delle quali è quella di trasformare la tensione alternata in tensione continua; ciò si realizza grazie al fatto che il flusso di corrente è solo unidirezionale, e quindi vengono preservate le sole semionde (p.es.) positive, tagliando quelle negative.

3.2.2 Il triodo

Noi non siamo interessati direttamente al diodo, ma alla sua diretta derivazione, cioè il *triodo*. Se tra anodo e catodo si interpone un terzo elettrodo, denominato *griglia* (figura 3.6a), costituito da un filo metallico molto sottile e avvolto a spirale intorno al catodo, il flusso di corrente anodica può essere controllato dal potenziale (negativo) con il quale viene caricata la griglia. Il fenomeno è ben evidente nella figura 3.7, nella quale si vede che se la griglia supera, in negativo, la cosiddetta *tensione di interdizione*, si crea una barriera negativa di potenziale (figura 3.7a) che blocca il flusso della corrente (zona tratteggiata). Nella figura 3.7b si vede invece la distribuzione del potenziale esattamente alla tensione di interdizione (in questo caso $-12\ V$), mentre aumentando in senso positivo la tensione di griglia (portandola p.es. a $-6\ V$) il potenziale positivo riesce a trapassare il piano della stessa, consentendo di attirare gli elettroni (3.7c) e attivando quindi un flusso di corrente. La possibilità, da parte della



(a) Struttura interna di un triodo

(b) Principio di funzionamento del triodo

Figura 3.6: Struttura e principio di funzionamento del triodo a vuoto

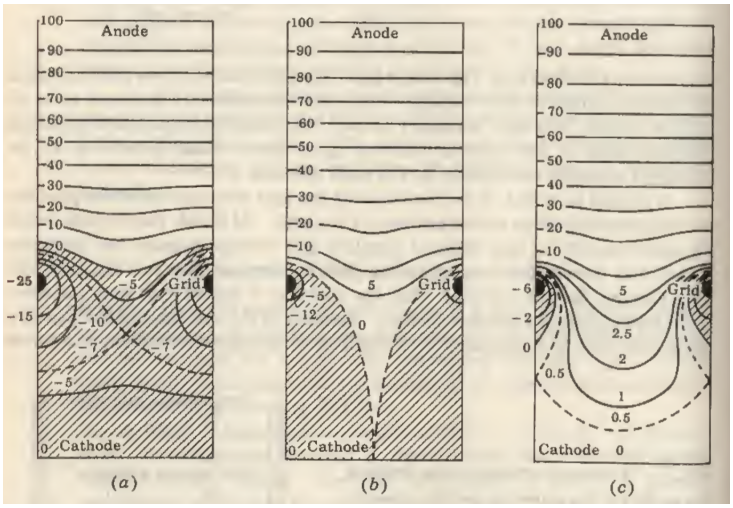


Figura 3.7: Contorno equipotenziale in Volts sul piano degli elettrodi di un triodo. In (a) la griglia è alimentata con una tensione inferiore a quella di interdizione ($V_g = -25\ V$); in (b) esattamente al valore di interdizione ($V_g = -12\ V$); in (c) a metà del valore di interdizione ($V_g = -6\ V$)

griglia, di consentire o bloccare un flusso di corrente usando solo piccole variazioni del proprio potenziale negativo, implica che una piccola variazione della tensione di griglia (che per altro è molto vicina al catodo), moduli una grossa variazione della corrente anodica; da ciò nasce l'attitudine del tubo termoionico ad *amplificare* i segnali (ma di ciò si parlerà nei corsi di Elettronica).

3.2.3 Curve caratteristiche del triodo

Se riportiamo su un diagramma la variazione della corrente anodica in funzione della tensione anodica, per valori costanti della tensione di griglia, si ottengono le cosiddette *curve caratteristiche* del triodo, rappresentate in figura 3.8. Si può osservare che, fissata una certa tensione di griglia (p.es. $V = -2,0\ V$), l'andamento qualitativo della curva è simile a quello del diodo, rappresentato in figura 3.5. L'equazione in questo caso dipende da entrambe le tensioni V_g di griglia e V_a anodica; quest'ultima incide sulla corrente tramite il coefficiente μ, detto *fattore di amplificazione*, che è sostanzialmente costante e indipendente dalla corrente.

$$I_a = k(V_g + V_a/\mu)^{3/2} \tag{3.2}$$

Se scendiamo con la V_g su valori molto negativi (p.es. $< -3,5V$ nel caso del tubo associato al diagramma di figura 3.8) per ottenere una certa corrente anodica (p.es. $0,5\ mA$) dobbiamo aumentare in modo molto significativo la tensione anodica. Se invece manteniamo costante la tensione anodica, ci sarà una certa tensione V_{int} per la quale si ha l'interdizione.
Per comprendere bene il funzionamento del triodo bisogna polarizzarlo correttamente e vedere cosa succede variando i parametri. Nel circuito di figura 3.9a il triodo è alimentato da una tensione E tramite la *resistenza di carico*. La somma della tensione V_a e di quella ai capi della resistenza deve equilibrare la tensione E. Se la tensione V_a va a zero (p.es. a seguito di un cortocircuito tra gli elettrodi del tubo) la corrente anodica vale $I_a = V_a/R$. Questa condizione di funzionamento corrisponde al punto all'estrema sinistra sulla retta gialla di figura 3.9b, sull'asse delle ordinate, in corrispondenza di $I_a = 3\ mA$ (in questo caso si è assunto che sia $R = 100\ k\Omega$). Se invece spegniamo il filamento del tubo, lo stesso smette di funzionare, non scorre più corrente, la caduta di tensione sulla R è nulla e tutta la tensione E di alimentazione si presenta ai capi del triodo, cioè $V_a = E$. Questa condizione di funzionamento corrisponde al punto all'estrema destra sulla retta gialla di figura 3.9b, sull'asse delle ascisse, in corrispondenza di $V_a = 300\ V$.

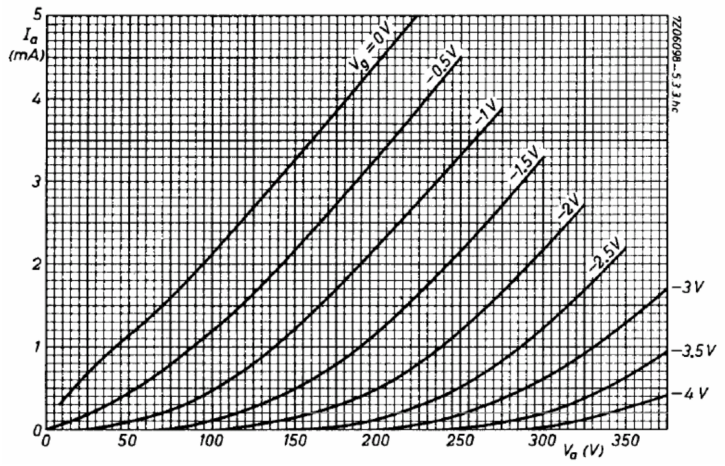
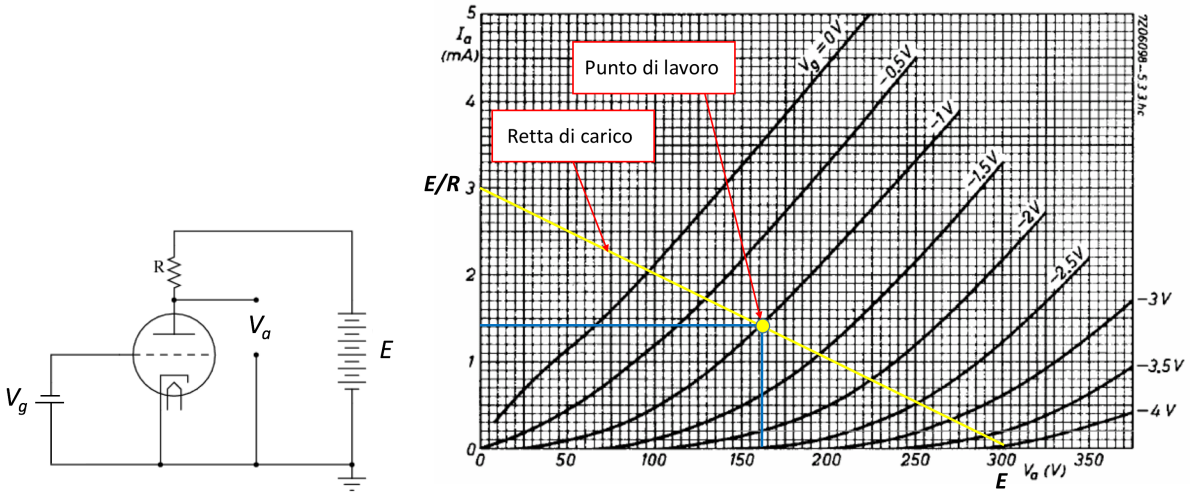


Figura 3.8: Curve caratteristiche del triodo



(a) Circuito di polarizzazione del triodo

(b) Punto di lavoro del triodo

Figura 3.9: Circuito di polarizzazione e punto di lavoro del triodo

Qualunque altro punto di lavoro intermedio tra questi due estremi giace sulla retta gialla, chiamata *retta di carico*. Per variare il punto di lavoro basta variare la tensione di griglia; se usiamo il valore $V_g = -1\ V$, il punto di lavoro si trova intersecando la curva per $V_g = -1\ V$ costante con la retta di carico, come si vede in figura 3.9b; per questa condizione si può leggere direttamente sul diagramma la tensione e la corrente anodica; i valori sono rispettivamente $V_a = 160\ V$ e $I_a = 1,4\ mA$.
Il funzionamento del triodo consente di controllare la tensione e la corrente anodica agendo solamente sulla tensione di griglia la quale, essendo polarizzata negativamente, non assorbe corrente; dunque si ha sempre $I_g = 0$. Ciò significa che possiamo controllare il flusso di una corrente (anodica) senza che ci sia assorbimento di corrente nel circuito di controllo (il circuito di griglia) e quindi senza che si dissipi potenza. Questo costituisce un salto di qualità enorme rispetto ai relè, che invece richiedono un'ingente dissipazione di potenza nel circuito di eccitazione. Un altro vantaggio rilevante è la velocità di commutazione, che è più bassa di alcuni ordini di grandezza rispetto a quella dei relè; qua non c'è nulla di elettromeccanico, non c'è alcuna massa da spostare per attivare l'eccitazione e

i tempi di commutazione sono legati alla sola geometria del tubo, alle tensioni in gioco e agli altri parametri dello stesso.
Come anticipato precedentemente, grazie al fattore di amplificazione μ dell'equazione 3.2, il triodo ha una naturale propensione ad amplificare i segnali alternati presenti sulla griglia; se infatti torniamo alla figura 3.9b e immaginiamo di variare la V_g nell'intorno del punto di lavoro, p.es. tra 0 e $-2\,V$, la tensione anodica varierà tra 97 e $223\,V$; ciò significa che una variazione di $2\,V$ in ingresso determina una variazione di $223 - 97 = 126\,V$ in uscita; in questo senso il triodo è un *amplificatore* di segnali. Questa funzionalità del triodo viene sfruttata nell'elettronica analogica, laddove ci sia appunto da amplificare segnali analogici. Nel nostro caso dobbiamo invece usare il triodo come interruttore comandato; questa funzionalità è evidenziata nella figura 3.10 ed è qualla associata alla logica Booleana.

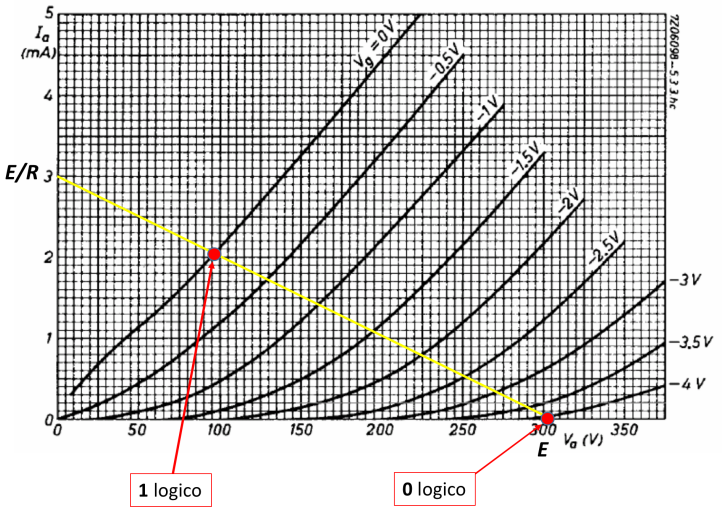


Figura 3.10: Triodo usato secondo la logica binaria

Se la griglia viene posta al potenziale di $0\,V$, la corrente anodica corrispondente è pari a circa $2\,mA$; questa è una condizione di piena conduzione, che possiamo associare alla variabile logica 1. Se invece imponiamo $V_g = -4\,V$ portiamo il triodo in interdizione, la corrente anodica si annulla e questa condizione viene associata alla variabile logica 0. Ovviamente la corrispondenza è del tutto convenzionale (si potrebbe invertire 0 con 1) e la si potrebbe attuare anche rispetto alla tensione anodica, in modo che, p.es., la condizione di interdizione caratterizzata da $V_a = 300\,V$ corrisponda a 1, mentre la condizione di piena conduzione caratterizzata da $V_a = 97\,V$ corrisponda a 0.
Subito dopo i primi computer elettromeccanici, che furono costruiti usando i relè, la tecnologia dei tubi a vuoto prese il sopravvento e durò fino ai primi anni $'60$. Ricordiamo che l'ENIAC, derivato dal primo computer sperimentale a tubi termoionici (l'*ABC*" di Atanasov e Berry), conteneva 17468 tubi.
I tubi termoionici, pur essendo di gran lunga migliori dei relè, hanno diversi difetti; sono ingombranti, fragili, e richiedono una potenza notevole per il riscaldamento del filamento e del catodo. Nei primi anni $'50$ si stava però preparando la tecnologia dei semiconduttori, che avrebbe trasformato completamente il mondo dei computer.

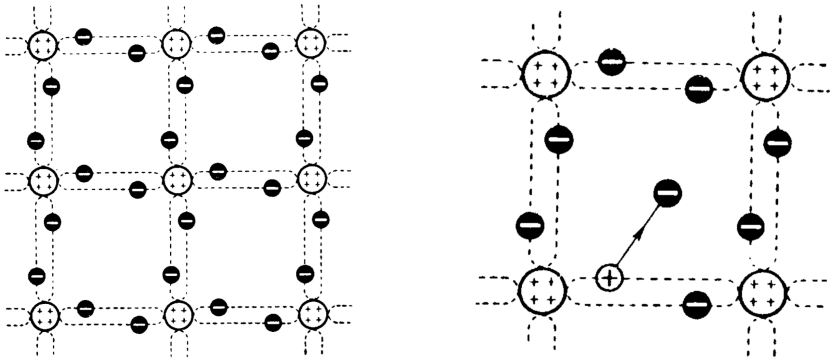
3.3 Il bit allo stato solido: il transistor

Nel 1948, nei laboratori *Bell* nasceva un piccolo dispositivo elettronico chiamato *transistor*, destinato a rivoluzionare lo sviluppo dell'elettronica. Così come il triodo può farsi derivare, storicamente e tecnicamente, dal diodo, anche il transistore trae le proprie origini dal *diodo a cristallo*, elemento circuitale entrato nella tecnica

elettronica negli anni della seconda guerra mondiale. Il funzionamento dei transistori e dei diodi è basato sulla conduzione elettrica dei cristalli *semiconduttori*. L'impiego dei semiconduttori nella tecnica elettronica risale ai primi del 900, poiché i cristalli di *galena* (solfuro di piombo) furono impiegati per le prime radio, mentre il selenio e l'ossido di rame vennero impiegati per realizzare i cosiddetti *raddrizzatori a secco*. Tuttavia è solo dopo gli esperimenti alla Bell che si sviluppò il transistor come lo conosciamo oggi.
E' sempre stato noto che, prendendo come elemento di raffronto la resistività, i semiconduttori stanno, per così dire, in mezzo fra conduttori e isolanti, essendo più vicini ai primi che non ai secondi (da ciò il nome di semiconduttori). Altrettanto nota è la proprietà che, a differenza dei metalli, essi hanno una resistività che diminuisce all'aumentare della temperatura; ben poco era però noto sul meccanismo di conduzione unilaterale presentata dai semiconduttori in determinate condizioni. Sappiamo ora che la difficoltà principale nello studio del comportamento dei semiconduttori sorgeva dal fatto che tracce anche minime di impurità hanno effetti rilevanti sulle proprietà degli stessi. Tali difficoltà furono superate sostituendo la galena col *germanio* e col *silicio* nei rivelatori a cristallo che nella seconda guerra mondiale furono usati nei radar. Il silicio e il germanio hanno la possibilità di essere ottenuti con un grado estremo di purezza, così che su di essi si è potuto sperimentare con rigore la teoria della conduzione elettrica nei semiconduttori formulata precedentemente. Alcune nozioni essenziali di questa teoria, pienamente sufficienti per la comprensione del funzionamento dei diodi e dei transistori, possono essere espresse in forma elementare; a esse dedicheremo i prossimi paragrafi.

3.3.1 Struttura dei semiconduttori

Il germanio (Ge) è un elemento tetravalente, del gruppo dello stagno, che ha numero atomico 32. Allo stato puro esso si presenta in forma policristallina, ma con particolari processi termici può essere ottenuto sotto forma di grossi cristalli isolati. Ciascun atomo è legato mediante legami covalenti a quattro atomi vicini (1, 2, 3, 4), posti ai vertici di un tetraedro regolare, in maniera tale che la distanza fra due qualunque dei cinque atomi è sempre la medesima (si veda la figura 3.11a). L'atomo di germanio è formato da un nucleo positivo e da 32 elettroni; il



(a) Atomi di germanio con i legami covalenti nel reticolo cristallino

(b) Fuoriuscita di un elettrone da un legame covalente e nascita di una lacuna

Figura 3.11: Legami covalenti e nascita di una lacuna in un cristallo di germanio

nucleo e 28 elettroni formano la parte inerte dell'atomo mentre i quattro elettroni rimanenti, cioè gli elettroni di valenza dell'orbita esterna, sono i responsabili dell'attività chimica ed elettrica dell'atomo e producono il legame con gli atomi vicini del reticolo cristallino. La configurazione degli atomi nel cristallo di germanio non è statica: la presenza di energia termica causa un'incessante vibrazione degli atomi del reticolo attorno alle loro posizioni di equilibrio. In conseguenza di ciò, già a temperatura ordinaria, alcuni elettroni di valenza possono acquistare energia sufficiente per rompere il rispettivo legame covalente, cioè per svincolarsi dal complesso di forze che li tengono avvinti al reticolo; tali elettroni rimangono liberi nello spazio vuoto interno al cristallo (enorme rispetto a quello occupato dagli atomi) e si muovono in maniera del tutto disordinata e casuale, come le molecole di un gas, senza sentire alcun effetto elettrico da parte degli atomi circostanti.

Se al cristallo è applicato un campo elettrico esterno, al moto casuale (moto termico) degli elettroni liberi si sovrappone una loro migrazione complessiva verso l'elettrodo positivo, così che ha luogo entro il cristallo una vera e propria corrente elettronica. Alla fuoruscita di ogni elettrone dal legame che lo teneva vincolato al reticolo corrisponde un altro importante fenomeno; nel posto lasciato libero dall'elettrone uscito si viene a manifestare un intenso campo elettrico (prima neutralizzato dalla presenza dell'elettrone) che tende ad attirare un nuovo elettrone nel vuoto formatosi: potrà essere catturato l'elettrone sfuggito o un altro elettrone libero, ma è più comune il fatto che il posto vuoto sia occupato da un elettrone di valenza di un atomo adiacente, quando l'agitazione termica lo porta in condizione favorevole per essere catturato. Con ciò la situazione non è sanata perché lo stato di squilibrio elettrico si è trasferito al nuovo atomo che ha perso un elettrone; si ripete allora il meccanismo di cattura di un nuovo elettrone da un atomo adiacente e così via. In tal modo, il posto lasciato vuoto dall'elettrone sfuggito per effetto termico si sposta entro il cristallo: lo spostamento è del tutto casuale e avviene unicamente sotto l'effetto della vibrazione termica degli atomi, ma assume una direzione e un verso preferenziale se il cristallo è sottoposto a un campo elettrico esterno che facilita la cattura in una direzione e in un verso piuttosto che negli altri. Poiché la mancanza di un elettrone in una regione inizialmente neutra equivale alla presenza di una carica positiva, lo spostarsi del posto vuoto equivale allo spostarsi di una carica positiva esattamente uguale e opposta a quella negativa dell'elettrone; a tale carica fittizia positiva diamo il nome di *lacuna*. Con tale convenzione possiamo allora dire che quando, sotto l'effetto dell'energia termica, si verifica la rottura di un legame covalente, si liberano nel cristallo un elettrone e una lacuna (si veda figura 3.11b), con cariche uguali e opposte, che si muovono liberamente entro il cristallo. Il moto è di per sè disordinato (moto termico), ma se il cristallo è sottoposto a un campo elettrico, acquista il carattere di migrazione ordinata (nel verso del campo per le lacune e nel verso opposto per gli elettroni), cioè di una corrente nel verso del campo.
Gli elettroni e le lacune prodotti per effetto termico non sussistono indefinitamente nel cristallo: infatti ogni elettrone libero, muovendosi casualmente entro il cristallo, finisce per cadere nel campo elettrico prodotto dalla fuoruscita di un altro elettrone. Esso sparisce perciò come elettrone libero e contemporaneamente sparisce la carica positiva corrispondente al posto vuoto: in definitiva si annulla contemporaneamente una lacuna. Si trova che in condizioni ordinarie la vita media di un elettrone (e corrispondentemente di una lacuna) è dell'ordine di un centinaio di microsecondi. Fra il numero di coppie di elettroni-lacune che si formano e quello delle coppie che si estinguono nasce uno stato di equilibrio per cui, a una determinata temperatura, il numero di coppie presenti entro il cristallo è mediamente costante; a temperatura ambiente esso è dell'ordine di $25 \cdot 10^{12}$ per cm^3 e cresce fortemente con la temperatura.
La presenza delle coppie libere entro il cristallo provoca in esso una certa conducibilità intrinseca: essa è legata al numero di copie elettroni-lacune presenti e dipende perciò dalla temperatura. Si tratta di una conducibilità molto modesta: la resistività corrispondente, a temperatura ambiente, ha un valore di $\rho = 0,5\,\Omega m$, pari a circa 30 milioni di volte quella del rame. Essa diminuisce fortemente al crescere della temperatura.
La formazione di coppie elettrone-lacuna può avvenire, oltre che per effetto termico, anche per altre cause, di cui fondamentali sono l'illuminazione e l'introduzione di atomi estranei (impurità chimiche). La luce che incide sul cristallo può, coi suoi fotoni, fornire energia sufficiente per lo sradicamento di elettroni di valenza dei legami che li vincolano al reticolo: hanno allora origine delle lacune, che si comportano in maniera identica a quelli prodotti per effetto termico; il fenomeno viene sfruttato nei cosiddetti *fotodiodi* e *fototransistor*. Della nascita di coppie di elettroni e lacune per effetto di impurità chimiche, argomento di fondamentale importanza per l'attuazione dei diodi e transistori, diremo nel prossimo paragrafo.
Le proprietà viste per il germanio valgono qualitativamente anche per il silicio, elemento tetravalente di numero atomico 14; in esso occorre però maggiore energia per rompere i legami covalenti. Ne deriva che, a temperatura ordinaria, il numero di coppie di elettrone-lacuna presenti è minore che nel germanio e pertanto la sua resistività intrinseca risulta ancora maggiore di quella del germanio, pari a circa $\rho = 640\,\Omega m$.

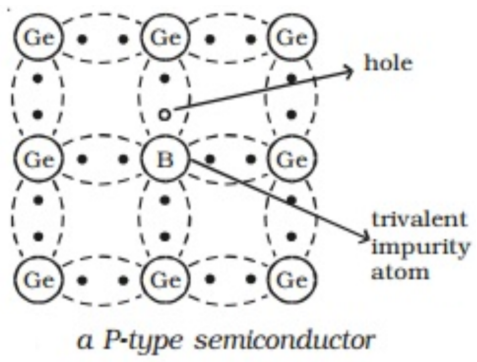
3.3.2 Semiconduttori di tipo *n* e di tipo *p*.

Tracce anche minime di impurità alterano fortemente la conducibilità del germanio; è particolarmente interessante, per la sua applicazione nei transistori, l'effetto che si ha quando l'impurità è costituita da un elemento tri-

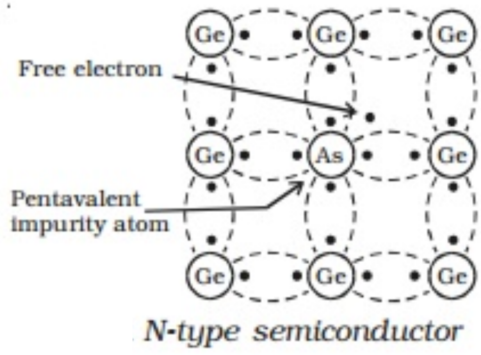
valente (boro, gallio, indio) o pentavalente (fosforo, arsenico, antimonio), da elementi, cioè, i cui atomi posseggano rispettivamente tre o cinque elettroni di valenza invece dei quattro del germanio.

Figura 3.12: Particolare della tavola periodica degli elementi che interessano i dispositivi semiconduttori

Se tracce di queste sostanze sono mescolate al germanio fuso (ad esempio, in ragione di un atomo estraneo per ogni milione di atomi di germanio), nella cristallizzazione avviene che gli atomi estranei entrano a far parte del reticolo cristallino in maniera identica agli atomi di germanio. Essendo pochissimi rispetto agli atomi di germanio, gli atomi estranei sono normalmente circondati completamente da atomi di germanio. Ciò posto, consideriamo il caso di un atomo estraneo pentavalente, per esempio l'arsenico (As) (si veda la figura 3.13b): i quattro atomi di germanio più vicini formano quattro legami covalenti con quattro dei suoi cinque elettroni di valenza. I quattro elettroni entrano così a far parte della struttura reticolare con legami identici a quelli degli atomi di germanio. Il quinto elettrone di valenza, invece, che non partecipa ai legami covalenti, con una minima energia (sempre presente a temperatura ordinaria) si svincola dall'atomo estraneo e diviene libero di muoversi entro il cristallo (nello stesso modo degli elettroni nati per effetto termico) contribuendo così alla sua conducibilità. L'atomo estraneo, che ha perso il suo quinto elettrone di valenza, diviene uno ione positivo, vincolato al reticolo dai legami covalenti coi quattro atomi di germanio che lo circondano: essendo immobile esso non contribuisce in alcun modo alla conducibilità del cristallo. Nel caso di impurità trivalenti, per esempio il boro (B) (si veda la figura 3.13a), poichè



(a) Cristallo di germanio drogato p



(b) Cristallo di germanio drogato n

Figura 3.13: Impurità di tipo n e p all'interno di un cristallo di germanio

ogni atomo estraneo che entra a far parte del reticolo cristallino ha tre soli elettroni di valenza, uno dei quattro legami covalenti che l'uniscono ai quattro atomi di germanio che lo circondano rimane incompleto per assenza di un elettrone. Ciò crea uno squilibrio elettrico, cui corrisponde un intenso campo elettrico, il quale finisce per catturare un elettrone da qualche atomo di germanio adiacente. Quando ciò avviene, l'atomo estraneo (trivalente) viene a possedere un elettrone in più e diviene uno ione negativo che rimane a far parte stabilmente del reticolo

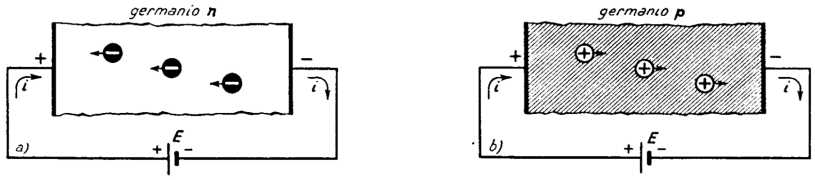
cristallino; il posto lasciato vuoto dall'elettrone catturato si comporta come una carica positiva che si sposta entro il cristallo, cioè è una lacuna che ha il medesimo comportamento delle lacune nate per effetto termico nel germanio puro. Essa contribuisce ad aumentare la conducibilità del cristallo; nessun effetto, invece, ha sulla conducibilità lo ione negativo, che è immobile entro il reticolo.
Dunque, la presenza di atomi estranei pentavalenti dà origine ad altrettanti elettroni liberi, mentre la presenza di atomi estranei trivalenti dà origine ad altrettante lacune. Il numero di elettroni nel primo caso e di lacune nel secondo, generati dalla presenza di tracce anche minime di impurità, è molto maggiore del numero degli elettroni e lacune generati a temperatura ordinaria per effetto termico; pertanto, in presenza di impurità, la conducibilità nel germanio è enormemente maggiore della conducibilità intrinseca, ed è quasi esclusivamente dovuta al movimento di elettroni nel caso di impurità pentavalenti e di lacune nel caso di impurità trivalenti.
I cristalli di germanio puro, cui siano state aggiunte determinate quantità di impurità pentavalenti o trivalenti, sono alla base del funzionamento dei diodi e dei transistori. Il germanio *drogato* (così si usa dire!) con atomi pentavalenti si suole chiamare germanio di tipo *n*, per indicare che la conducibilità è prevalentemente dovuta a cariche negative (elettroni); il germanio drogato con atomi trivalenti si dice di tipo *p*, per indicare che la conducibilità è dovuta a cariche mobili positive.
La conducibilità dei due tipi di germanio dipende dalla percentuale di atomi esterni introdotti nel reticolo cristallino: mentre nel germanio puro, a temperatura ordinaria, è come abbiamo visto di $\rho = 0,5\ \Omega m$, con un drogaggio debole (1 atomo estraneo di fronte a 250 milioni di atomi di germanio) la resistività del germanio diviene $\rho = 0,1\ \Omega m$. Con un drogaggio moderato (un atomo estraneo ogni 14 milioni di atomi di germanio) la resistività scende a $\rho = 10^{-2}\ \Omega m$, mentre si abbassa a $\rho = 10^{-4}\ \Omega m$ nei cristalli fortemente drogati (un atomo estraneo ogni 250000 di germanio); tanto per avere un'idea di questi valori, si tenga presente che la resistività del carbone, usato per le spazzole delle macchine elettriche, è dell'ordine di $\rho = 0,2 \div 1 \cdot 10^{-4}\ \Omega m$.

Ge puro	Ge drogato 1 atomo su 250 mln	Ge drogato 1 atomo su 14 mln	Ge drogato 1 atomo su 250000
$0.5\ \Omega m$	$0.1\ \Omega m$	$10^{-2}\ \Omega m$	$10^{-4}\ \Omega m$

Figura 3.14: Tabella delle resistività del germanio per diversi livelli di drogaggio.

3.3.3 Meccanismo della conduzione nei semiconduttori di tipo *n* e di tipo *p*

Analizziamo ora il meccanismo con cui avviene il passaggio di corrente in un semiconduttore (germanio o silicio) drogato di tipo *n* o di tipo *p*, collegato esternamente mediante due elettrodi metallici (si veda la figura 3.15). Consideriamo dapprima il conduttore ottenuto con germanio di tipo *n*, indicato nella figura 3.15a. Ricordia-



(a) Conduzione in un cristallo di germanio drogato *n*

(b) Conduzione in un cristallo di germanio drogato *p*

Figura 3.15: Meccanismo di conduzione di un cristallo di germanio drogato *p* o *n*

mo che entro il cristallo vi sono elettroni liberi (simboleggiati in figura dai cerchietti neri), staccatisi dagli atomi estranei pentavalenti; essi si muovono di moto termico fra gli atomi di germanio e gli atomi pentavalenti (ionizzati positivamente) fissati al reticolo cristallino. Se, mediante una tensione E applicata agli elettrodi si crea un campo elettrico, si produce un moto ordinato di elettroni che si spostano respinti dall'elettrodo negativo ed attirati da quello positivo. Man mano che gli elettroni giungono nelle vicinanze dell'elettrodo positivo, essi vengono catturati

da questo e immessi nel circuito esterno: contemporaneamente nel cristallo, vicino all'elettrodo negativo, viene a crearsi una regione vuota di elettroni, nella quale predomina quindi la carica positiva degli ioni vincolati al cristallo (non più controbilanciata dalla carica distribuita degli elettroni liberi). Tale carica esercita un effetto attrattivo sugli elettroni presenti nell'elettrodo negativo: questi, pertanto, dal circuito esterno penetrano nel cristallo, in misura tale da controbilanciare la fuoruscita di elettroni che si ha dall'altra parte verso il circuito esterno. In definitiva si ha un continuo flusso di elettroni, cioè una corrente elettronica, sia attraverso il cristallo che nel circuito esterno e quindi una corrente i che ha il verso convenzionale indicato in figura.
Diversa è la situazione nel conduttore attuato con semiconduttore di tipo *p*, indicata nella figura 3.15b; in esso gli atomi trivalenti estranei si sono impossessati ciascuno di un elettrone appartenente ad atomi di germanio, divenendo ioni negativi. Il posto lasciato vuoto dall'elettrone sottratto viene occupato da un elettrone strappato ad un altro atomo e così via. Dunque, mentre gli ioni negativi estranei e gli atomi di germanio sono immobili nel reticolo, il posto vuoto si sposta casualmente come fosse un elettrone libero positivo (lacuna). In presenza di tensione E applicata ai due elettroni metallici, il moto delle lacune acquista un carattere preferenziale verso l'elettrodo negativo; da questo le lacune attirano elettroni e con essi si combinano neutralizzandosi. Contemporaneamente, dalla parte opposta del cristallo, l'elettrodo positivo strappa elettroni nelle immediate adiacenze, dando origine ad altrettante lacune che si spostano verso l'elettrodo negativo, e così via. In definitiva, quindi, dalla parte dell'elettrodo negativo entrano nel cristallo elettroni che neutralizzano le lacune; dall'altra parte vengono strappati elettroni dal cristallo (ed immessi nel circuito esterno) con conseguente formazione di lacune. Nel circuito esterno vi è ancora (e non può essere altrimenti) una corrente elettronica come nel caso della figura 3.15b, mentre entro il cristallo vi è un moto di cariche positive: il verso convenzionale della corrente i che scorre nel circuito coincide col senso del moto delle lacune.
Per completare il quadro della conduzione nei cristalli drogati *p* e *n* occorre tenere conto della conduzione intrinseca; indipendentemente dall'esistenza delle impurità, nel cristallo si ha la formazione di un certo numero di coppie elettrone-lacuna per effetto termico. Perciò nel germanio *n*, oltre agli elettroni (cariche mobili, o portatori di maggioranza) vi sono sempre anche lacune (portatori di minoranza) che partecipano, sia pure in misura assai minore, alla conduzione. Analogamente nel germanio *p*, se è vero che i portatori di maggioranza sono le lacune, vi sono anche degli elettroni (portatori di minoranza) che partecipano alla conduzione. A temperatura ordinaria i portatori di minoranza hanno effetto scarso sulla conduzione (seppure non trascurabile), ma acquistano importanza sempre maggiore al crescere della temperatura perché il numero di coppie generate per effetto termico cresce con la temperatura.

3.3.4 Diodo a giunzione *p-n*

Se due cristalli di germanio (o di silicio), uno di tipo *p* e l'altro di tipo *n*, sono portati a contatto intimo fra loro, ha luogo un fenomeno interessante: alcune lacune del germanio *p* e alcuni elettroni del germanio *n*, nel loro incessante moto caotico, attraversano la giunzione - cioè la superficie di contatto fra i due tipi di germanio - continuando a muoversi dalla parte opposta. I due cristalli di germanio, che originariamente erano allo stato neutro, tendono ora ad assumere una carica diversa: il germanio *p* (Ge-*p*) acquista una carica negativa, perché perde lacune che penetrano nel germanio *n* e riceve elettroni da questo; il germanio *n* (Ge-*n*) acquista invece una carica positiva, perché perde elettroni e riceve lacune dal Ge-*p* (si veda la figura 3.16). Con questo processo automatico di carica, però, si viene a creare una barriera di potenziale attraverso la giunzione che si oppone al moto delle cariche attraverso essa: infatti il Ge-*n* (che si porta a potenziale positivo rispetto a quello *p*) respinge le lacune del Ge-*p* che tendessero ad avvicinarsi alla giunzione, mentre il Ge-*p*, che diviene negativo, respinge gli elettroni che dal Ge-*n* tendessero ad attraversare la giunzione. In definitiva, fra i due cristalli di germanio si stabilisce a regime una differenza di potenziale tale che ogni corrente attraverso la giunzione è resa impossibile: il complesso dei due cristalli di germanio è divenuto qualcosa di analogo ad un condensatore carico.
Connettiamo ora due elettrodi metallici ai cristalli di germanio della giunzione e applichiamo una differenza di potenziale fra loro (figura 3.17). Se la polarizzazione è tale da aumentare la barriera di potenziale esistente fra il germanio *p* e quello *n* (*polarizzazione inversa*), nessuna corrente è possibile attraverso alla giunzione. Se invece la differenza di potenziale applicata è tale da eliminare la barriera di potenziale, il fluire di lacune dal germanio *p*

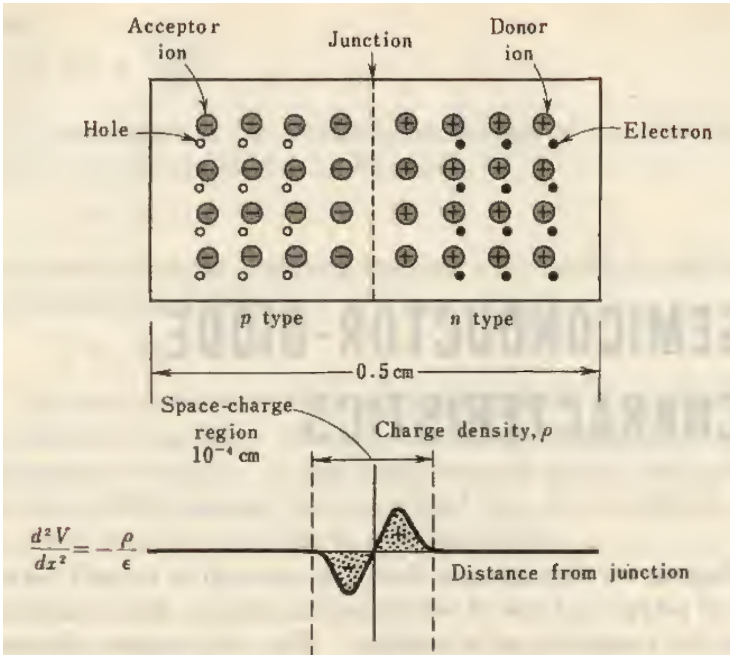


Figura 3.16: Barriera di potenziale di una giunzione *p-n*

al germanio *n* e degli elettroni dal germanio *n* a quello *p* non trova alcun ostacolo, e avviene con le caratteristiche viste precedentemente. Si ha dunque una corrente nel circuito che ha il verso convenzionale indicato nella figura 3.17b (corrente entrante nel Ge-*p* ed uscente dal Ge-*n*): se la tensione applicata è tale non soltanto da eliminare la barriera di potenziale ma addirittura da creare una differenza di potenziale in senso opposto (Ge-*p* positivo rispetto al Ge-*n*) il moto delle lacune e degli elettroni è agevolato, e la corrente ha valore maggiore. In base a ciò si deduce che la giunzione *p-n* ha proprietà conduttrici unidirezionali: essa lascia passare corrente (nel verso p-*n*) solo quando la tensione applicata è tale da rendere positivo il Ge-*p* rispetto al Ge-*n*; questa condizione è detta di *polarizzazione diretta*.

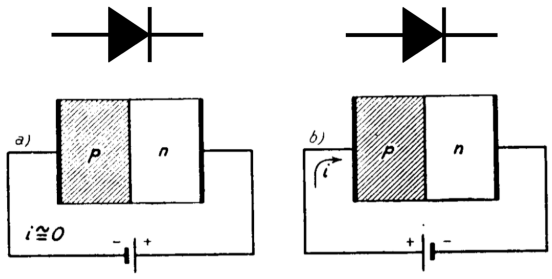


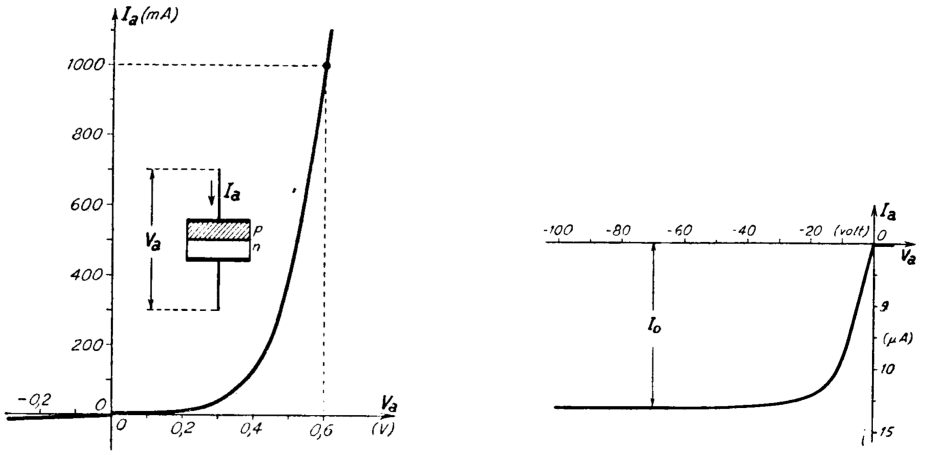
Figura 3.17: Diodo polarizzato inversamente (a) e direttamente (b)

L'esistenza di un verso preferenziale nel passaggio di corrente attraverso la giunzioni *p-n* è alla base del funzionamento dei diodi a semiconduttore. Tali dispositivi, grazie alla loro conduzione praticamente unilaterale, possono essere usati come raddrizzatori in alternativa dei tubi a vuoto. Come per questi, la conoscenza fondamentale delle proprietà dei diodi a giunzione si ottiene dall'esame della curva caratteristica, ottenuta determinando sperimentalmente il valore della corrente in funzione della tensione applicata fra gli elettrodi della giunzione.

Nella figura 3.18a è indicato l'andamento generale della caratteristica di un diodo commerciale al germanio, usato quale raddrizzatore per alimentatori: tale andamento è grosso modo assimile a quello della caratteristica di un diodo a vuoto, ma ne differisce sia per gli ordini di grandezza delle tensioni e delle correnti, sia per l'esistenza di una corrente inversa non nulla, ma anche per la diversa legge che lega tensione e corrente, che in questo caso è di tipo esponenziale

$$I_a = I_d \left(e^{\frac{V_a}{\eta V_T}} - 1 \right) \tag{3.3}$$

Nella relazione (3.3) I_d è la corrente di diffusione della giunzione, η è una costante che vale circa 1 per il germanio e 2 per il silicio, mentre V_T è una tensione equivalente dovuta alla temperatura, di valore pari a circa $26mV$. L'andamento della corrente inversa, mal deducibile dalla figura 3.18a , è riportato in scala più ampia nella figura 3.18b; la corrente è nulla quando è nulla la differenza di potenziale fra gli elettrodi, ma cresce al crescere della tensione inversa fino a raggiungere un valore massimo costante, detta *corrente inversa di saturazione* I_{C0}; essa corrisponde al moto di tutte le coppie elettrone-lacuna generate per effetto termico: il suo valore dipende fortemente dalla temperatura.



(a) Curva caratteristica di un diodo a giunzione

(b) Corrente inversa in un diodo a giunzione

Figura 3.18: Curva caratteristica e corrente inversa per un diodo a giunzione

Le proprietà raddrizzatrici del diodo a giunzione risultano evidenti osservando che la corrente diretta di figura 3.18a ha il valore di $1\ A$ per una tensione applicata di soli $0,6\ V$, mentre occorrono $10\ V$ di tensione inversa per ottenere una corrente di una decina di microampere, e anche con tensioni assai forti non si raggiungono $15\ \mu A$. I valori descritti sono particolari del diodo considerato, ma i rapporti fra corrente diretta ed inversa rimangono sostanzialmente i medesimi in ogni caso. Nei diodi al silicio, anzi, la corrente inversa è, in proporzione, ancora minore; per contro la sua dipendenza dalla temperatura è più sentita nei diodi al silicio che in quelli al germanio. La forte corrente diretta che i diodi a giunzione sono in grado di far passare con tensioni applicate piccolissime, rappresenta un vantaggio notevole rispetto ai diodi a vuoto, sfruttabile specialmente nel raddrizzamento delle correnti alternate.

3.3.5 La struttura del transistor

Supponiamo ora che nella giunzione *p-n* uno dei due elementi della stessa, ad esempio il germanio di tipo *n*, sia drogato molto più debolmente rispetto all'altro. Allora il numero di elettroni che si muovono all'interno della giunzione è molto minore di quello delle lacune, e si viene ad avere la singolare situazione in cui nel germanio *n* si muovono più lacune che elettroni: in tali condizioni si suole dire che si è avuta iniezione di lacune dal germanio

p al germanio *n*. Similmente si può fare in modo che sia il il germanio *p* a essere drogato molto più debolmente rispetto al germanio *n*. In tal caso si avrà un'iniezione di elettroni dal germanio *n* al germanio *p*. Nell'uno o nell'altro caso, se uno dei due elementi della giunzione è drogato debolmente, il numero delle sue cariche libere è piccolo e quindi è piccolo in proporzione anche il numero degli elettroni e lacune che incontrandosi fra loro si neutralizzano. Piccola, pertanto, risulta la corrente nel circuito esterno: in effetti, con un drogaggio molto leggero di uno dei due elementi della giunzione, è possibile mantenere le due parti allo stesso potenziale con una corrente molto esigua.
Il processo di emissione, con l'inevitabile, seppure piccolo, passaggio di corrente nel circuito di polarizzazione della giunzione, è fondamentale per il funzionamento del transistore: la parte di giunzione drogata debolmente si dice *base*, mentre si dice *emettitore* il cristallo più drogato. Se l'emettitore è un cristallo di tipo *p* esso inietta lacune nella base che è di tipo *n* (fig. 3.19.a); se l'emettitore è un cristallo di tipo *n*, esso inietta elettroni nella base di tipo *p* (fig. 3.19.b).

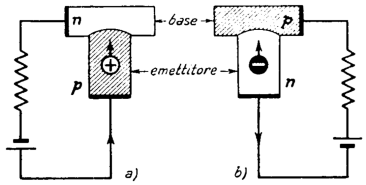


Figura 3.19: Base ed emettitore in due giunzioni *p-n* e *n-p*

Se sulla base viene saldato un secondo elemento di germanio dello stesso tipo di quello dell'emettitore, che si dice *collettore*, si realizza un *transistore*, che è un dispositivo a tre elettrodi; la costituzione schematica del transistore risulta pertanto quella indicata nella figura 3.20.

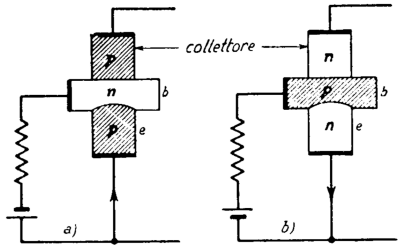
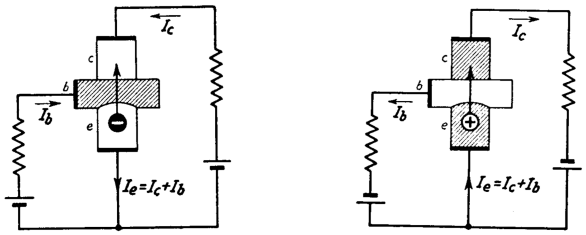


Figura 3.20: Transistor di tipo *pnp* e *npn*

Si hanno dunque, a seconda del tipo di germanio prescelto per l'emettitore, due tipi diversi di transistori che si denominano rispettivamente *pnp* (fig. 3.20a) e *npn* (fig. 3.20b).
Consideriamo ora il transistore di tipo *npn*; nell'ipotesi che, mediante una tensione esterna V_{be} tra base ed emettitore, la base sia mantenuta a un potenziale maggiore di quello dell'emettitore, gli elettroni di questo si muovono liberamente attraverso la giunzione *np*, si diffondono entro alla base e, attraverso questa, entrano nel collettore. Se questo venisse lasciato isolato (come in fig. 3.20b) esso si caricherebbe negativamente, a causa dell'apporto di elettroni, e ogni diffusione di questi dalla base cesserebbe. Se invece, com'è indicato nella figura 3.21a, il collettore è polarizzato positivamente rispetto all'emettitore (e di conseguenza alla base), gli elettroni penetrati nel collettore si dirigono verso l'elettrodo positivo e attraverso esso ritornano all'emettitore tramite il circuito esterno.

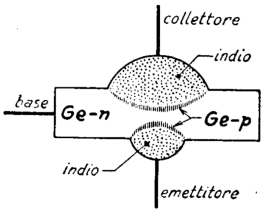


(a) Transistor di tipo *npn* correttamente polarizzato

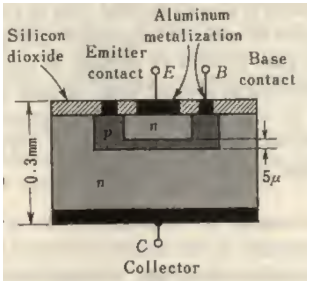
(b) Transistor di tipo *pnp* correttamente polarizzato

Figura 3.21: Polarizzazione dei transistor dei due tipi

Si ottiene in tal modo una corrente I_c nel circuito collettore-emettitore, il cui valore può essere anche assai ragguardevole, specialmente se confrontato con quello della corrente I_b, che si ha nel circuito di polarizzazione della base. La corrente I_c viene sfruttata nel funzionamento del transistore così come accade per la corrente anodica nei tubi elettronici; il suo valore può essere regolato con grande facilità agendo sulla differenza di potenziale fra base ed emettitore.
Se aumentiamo la tensione V_{be} tra base ed emettitore, un maggior numero di elettroni penetra nella base e diffondendosi attraverso essa entra nel collettore. Qui gli elettroni si dirigono verso l'elettrodo positivo e da questo entrano nel circuito esterno producendo un aumento della corrente.
Se, viceversa, la tensione V_{be} viene diminuita, diminuisce il numero di elettroni iniettati nella base e quindi la corrente nel collettore. In definitiva, come nel triodo si ha una corrente anodica agevolmente regolabile dalla tensione di griglia, così nel transistore si ha una corrente di collettore il cui valore è facilmente regolabile agendo sulla differenza di potenziale fra base ed emettitore. Occorre però tenere sempre presente che, a differenza di quanto accade normalmente per i triodi, nel circuito di regolazione base-emettitore circola una corrente I_b che provvede a reintegrare nella base le cariche positive (lacune) neutralizzate dagli elettroni in transito; il suo valore dipende, come quello di I_c, dalla differenza di potenziale fra base ed emettitore, ma è sempre molto più piccolo di I_c.
Quanto è stato detto per i transistori *npn* vale anche, coi dovuti cambiamenti di segno, per i transistori *pnp*. In questi (fig. 3.21b) il collettore deve essere polarizzato negativamente rispetto all'emettitore e alla base, per favorire il deflusso delle lacune dalla base al collettore; le lacune si dirigono allora verso l'elettrodo negativo del collettore, e ivi si neutralizzano a spese di elettroni che giungono dal circuito esterno. Sono tali elettroni che nel circuito esterno danno luogo alla corrente I_c (corrente di collettore) che naturalmente ha verso opposto a quella che si ha nei transistori *npn*; anche la corrente di base, che provvede a reintegrare nella base gli elettroni catturati dalle lacune in transito, ha segno opposto a quella che si ha nei transistori *npn*. L'azione di comando sulla corrente di collettore da parte della V_{be} si esplica nei transistori *pnp* come in quelli *npn*; più in generale, a parte i segni, non c'è una differenza essenziale di funzionamento fra i transistori dei due tipi. Non esiste, invece, per i transistor *pnp*, la possibilità di stabilire un'analogia coi triodi, a causa dei segni delle correnti e delle tensioni che sono opposte, dato che entro il transistore *pnp* si muovono cariche positive (lacune), mentre nel triodo si muovono solo elettroni.
Nella figura 3.22a si vede la struttura di un transistor realizzato secondo la tecnica più frequente per i transistor di piccole potenze, quella della *giunzione a lega*. Si parte da una piastrina di germanio, tagliata da un cristallo di tipo *n*, che costituirà la base; sulle due facce di essa si comprimono due dischetti di indio (elemento trivalente) e il tutto viene posto entro appositi forni, e portato a una temperatura tale che le gocce di indio fondano. L'indio fuso si dissolve parzialmente entro il germanio formando una lega con esso; arrestando al momento giusto questo processo si vengono a ottenere, fra la lega di indio e il germanio, due regioni di separazione, che risultano di tipo *p*. Sono tali regioni, insieme con la zona centrale di tipo *n*, che costituiscono il transistore; gli elettrodi sono poi saldati alle due gocce d'indio e a un fianco della piastrina di germanio. Le due gocce d'indio sono di dimensioni diverse, così che una delle regioni di tipo *p* risulta più vasta dell'altra; essa è usata per il collettore onde rendere più agevole la raccolta delle lacune che dall'emettitore giungono attraverso la base, resa sottilissima nella zona interposta fra emettitore e collettore. La figura 3.22b mostra invece la struttura di tipo *planare* per i transistor, usate per potenze maggiori e per la realizzazione dei transistor all'interno dei circuiti integrati.



(a) Struttura interna di un transistor di tipo *npn* costruito secondo il processo della giunzione a lega



(b) Realizzazione planare di un transistor

Figura 3.22: Due diverse tecniche di costruzione di un transistor

3.3.6 Curve caratteristiche di uscita del transistor

Anche per il transistor è possibile tracciare una serie di curve caratteristiche, che servono per capire il comportamento dello stesso e per dimensionare i vari circuiti di polarizzazione. Nei corsi di Elettronica si studieranno nel dettaglio tutte le possibili configurazioni (emettitore comune, base comune, ecc) e le soluzioni circuitali (push-pull, totem-pole, darlington...); per i nostri scopi è sufficiente capire il funzionamento di base.
Contrariamente al triodo, il cui circuito d'ingresso non assorbe corrente e dunque potenza, il circuito d'ingresso del transistor, dato dalla base, viene comandato in corrente; ecco allora che le curve d'uscita sono a corrente costante. Se riportiamo su un diagramma la variazione della corrente di collettore in funzione della tensione di collettore, per valori costanti della corrente di base, si ottengono le curve caratteristiche del transistor rappresentate in figura 3.23. Si può osservare che, fissata una certa corrente di base (p.es. $I_b = -50\ \mu A$), l'andamento qualitativo della

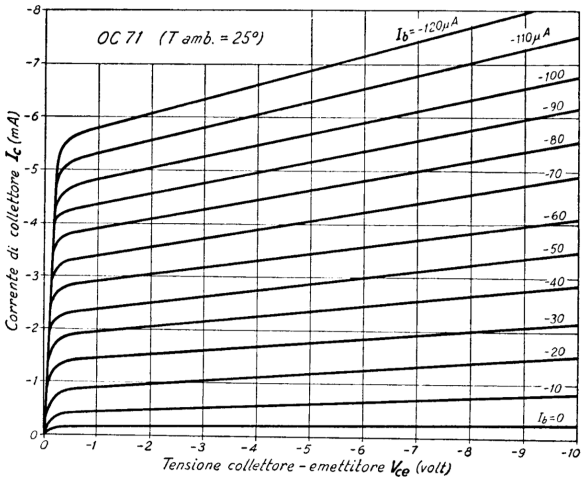
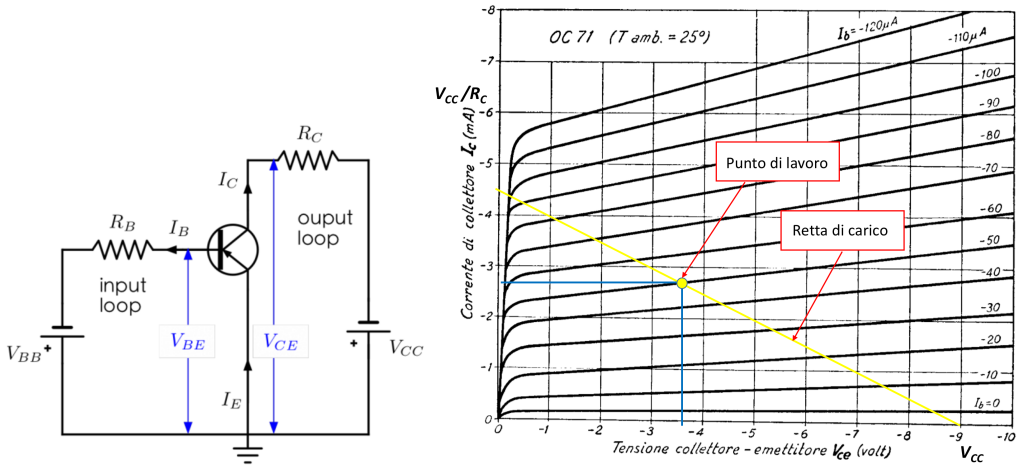


Figura 3.23: Curve caratteristiche di un transistor *pnp*

curva è piuttosto diverso da quello del triodo; la curva è data sostanzialmente dalla composizione di due rette, raccordate da una curva a gomito. Ciò significa che il transistor ha due zone di funzionamento ben distinte, quella prima del gomito, chiamata *regione di saturazione*, e quella dopo il gomito, che è la *regione attiva*, di normale impiego quando si usa il transistor come amplificatore.
Se scendiamo con la I_b su valori prossimi o uguali a $0V$, per ottenere una certa corrente di collettore (p.es. $0,2mA$) dobbiamo aumentare in modo molto significativo la tensione V_{CE} tra collettore ed emettitore. Se invece manteniamo costante la tensione V_{CE}, polarizzando debolmente in modo contrario la giunzione base-emettitore si arriva



(a) Circuito di polarizzazione di un transistor *pnp*

(b) Punto di lavoro del transistor

Figura 3.24: Circuito di polarizzazione e punto di un transistor *pnp*

all'interdizione, cioè al blocco della corrente di collettore.
A parte il fatto che il transistor è comandato in corrente, e che rispetto al triodo ha delle curve di forma diversa, per il resto il dimensionamento del circuito procede come per il triodo. Nel circuito di figura 3.24a il transistor è alimentato da una tensione V_{CC} tramite la *resistenza di carico* R_C. La somma della tensione V_{CE} e di quella ai capi della resistenza deve equilibrare la tensione V_{CC}. Se la tensione V_{CC} va a zero (p.es. a seguito di un cortocircuito tra gli elettrodi del transistor), la corrente di collettore vale $I_C = V_{CC}/R_C$. Questa condizione di funzionamento corrisponde al punto all'estrema sinistra sulla retta gialla di figura 3.24b, sull'asse delle ordinate, in corrispondenza di $I_C = -4,5\ mA$ (in questo caso si è assunto che sia $R_C = 2\ k\Omega$). Se invece polarizziamo inversamente la giunzione base-emettitore, non scorre più corrente sul circuito di uscita, la caduta di tensione sulla R_C è nulla e tutta la tensione V_{CC} di alimentazione si presenta ai capi del transistor, cioè $V_{CE} = V_{CC}$. Questa condizione di funzionamento corrisponde al punto all'estrema destra sulla retta gialla di figura 3.24b, sull'asse delle ascisse, in corrispondenza di $V_a = -9\ V$. Qualunque altro punto di lavoro intermedio tra questi due estremi giace sulla retta gialla, chiamata anche in questo caso *retta di carico*. Per variare il punto di lavoro basta variare la corrente di base; se usiamo il valore $I_b = -50\ \mu A$, il punto di lavoro si trova intersecando la curva per $I_b = -50\ \mu A$ costante con la retta di carico, come si vede in figura 3.24b; per questa condizione si possono leggere direttamente sul diagramma la tensione e la corrente di collettore; i valori sono rispettivamente $V_a = -3,6\,V$ e $I_a = -2,7\,mA$. Senza entrare nel merito delle equazioni di funzionamento del transistor, che sono alquanto complesse e che fanno riferimento al modello matematico di *Ebers-Mall*, possiamo solo dire che, così come succede per il triodo, una piccola variazione della grandezza d'ingresso - in questo caso la corrente I_B - determina una rilevante variazione della grandezza d'uscita, cioè la corrente di collettore. Il fenomeno è ben evidente dalla figura 3.24b: se spostiamo il punto di lavoro incrementando la corrente di base al valore $I_B = -70\ \mu A$, la corrente di collettore I_C varia da $I_a = -2,7\ mA$ a $I_a = -3,5\ mA$, determinando un rapporto $\Delta I_C/\Delta I_B$ pari a 35. In prima approssimazione, e limitatamente al caso dei grandi segnali, l'equazione che lega la I_C alla I_B è la seguente

$$I_C = \beta I_B + (1+\beta) I_{CB0} \tag{3.4}$$

dove I_{CB0} è la *corrente inversa di saturazione* tra base e collettore, mentre β rappresenta il *guadagno di corrente*. Il punto di lavoro del transistor illustrato in figura 3.24b è quello che si usa nell'elettronica analogica, quando si vuole impiegare il transistor come amplificatore. Nel nostro caso dobbiamo invece usare il transistor come interruttore comandato; questa funzionalità è evidenziata nella figura 3.25 ed è qualla associata alla logica Booleana.
Se la corrente di base supera il valore $I_B = -100\ \mu A$, allora la tensione V_{CE} diventa indipendente da I_B e si attesta su una valore praticamente costante V_{CEsat}, detto *tensione di saturazione* che vale circa $0,2 - 0,3\ V$ per il germanio e $0,6 - 0,7$ per il silicio. Questa è una condizione di piena conduzione, che possiamo associare alla

variabile logica 1.
Se invece imponiamo polarizziamo in modo debolmente inverso la giunzione base-emettitore, portiamo il transistor in interdizione, la corrente di collettore si annulla e questa condizione viene associata alla variabile logica 0. Anche in questo caso la corrispondenza è del tutto convenzionale (si potrebbe invertire 0 con 1) e la si potrebbe attuare anche rispetto alla tensione V_C, in modo che, p.es., la condizione di interdizione caratterizzata da $V_{CE} = -9\,V$ corrisponda a 1, mentre la condizione di saturazione caratterizzata da $V_{CE} = 0,2\,V$ corrisponda a 0.
Subito dopo i primi computer a tubi, tra cui il famoso ENIAC, i transistor presero rapidamente il sopravvento, visto che tra le due tecnologie esiste un rapporto di volumi e di potenze di diversi ordini di grandezza. Tanto per fare un paragone si pensi che l'ENIAC, il primo calcolatore interamente elettronico, funzionava grazie a 17468 tubi termoionici, equivalenti ad altrettanti transistor, e utilizzava $160\,kW$ di potenza elettrica praticamente solo per tenerli accesi (pari alla potenza richiesta da 80 forni elettrici di tipo domestico). Un moderno computer contiene miliardi di transistori e consuma più o meno un millesimo di quella potenza.
I transistor offrono in genere vantaggi enormi rispetto ai tubi termoionici, il cui impiego attuale è limitato ad alcune nicchie nell'ambito industriale, dove servono tensioni di lavoro di oltre $20kV$. Ma l'aspetto più rilevante è che grazie alla tecnologia planare, cui abbiamo fatto cenno in figura 3.22b, la tecnologia dei transistor apre la strada a quella dell'integrazione su larga scala, che porta ai circuiti integrati, costituiti da miliardi di transistor e altri componenti integrati su un unico chip di silicio.

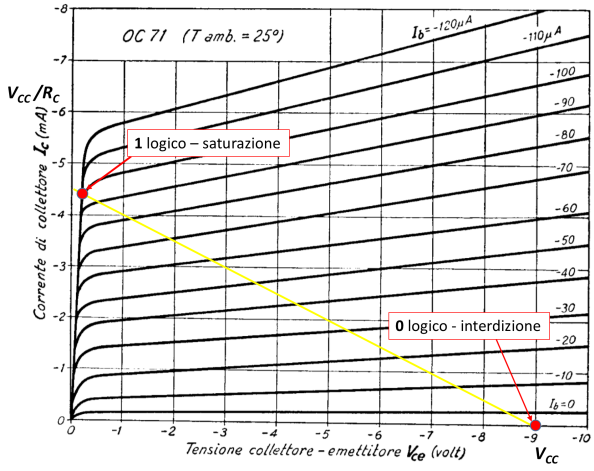


Figura 3.25: Transistor usato secondo la logica binaria

Capitolo 4

Algebra Booleana e porte logiche

4.1 Calcolo funzionale di verità

4.1.1 Introduzione

Come anticipato brevemente nel capitolo 1, la logica Booleana e le corrispondenti regole di manipolazione dei suoi elementi definite nell'*Algebra Boolena*, sono lo strumento concettuale chiave per lo sviluppo di *tutta* la tecnologia microelettronica che ha portato ai computer moderni, alle loro reti, a Internet, alla televisione digitale, alla telefonia mobile e, in generale, a tutti quei dispositivi elettronici di elaborazione dei segnali finalizzata al monitoraggio, controllo e gestione, locale e a distanza, degli strumenti automatici che usiamo quotidianamente. Quando usiamo il computer, lo smartphone, il telecomando per il cancello elettrico, quando attiviamo l'allarme di casa, usiamo un lettore ottico al supermercato, paghiamo col bancomat o freniamo in emergenza guidando l'automobile, stiamo usando l'Algebra Booleana.
Fatte queste premesse, è evidente che essa è la colonna portante di tutto lo sviluppo tecnologico della nostra società dell'informazione e che sia assolutamente necessario comprendere in profondità il suo funzionamento.
La Logica Booleana nasce nel 1854 a opera del matematico inglese George Boole, ma rimane inizialmente confinata nell'ambito degli specialisti di matematica. Solo quasi cent'anni dopo viene sfruttata a livello applicativo, quando nel 1938 Claude Elwood Shannon ne adattò il simbolismo all'analisi dei circuiti di commutazione, uno strumento essenziale nel progetto dei circuiti logici, che sono la base costitutiva dei moderni microprocessori e di tutta la circuiteria a essi asservita. L'algebra Booleana, nella sua declinazione binaria, contempla solo due valori logici atti a rappresentare il *vero* e il *falso* in una proposizione e ci permette di studiare formalmente i problemi della *logica deduttiva*. Successivamente, con Shannon, tali valori logici vengono associati allo stato di apertura o di chiusura di un generico contatto, oppure alla presenza o all'assenza di un segnale in un particolare punto di un circuito. I due valori logici, che sono mutuamente escludentisi, sono chiamati *costanti logiche* o *costanti binarie* e sono indicati di solito con i simboli 0 e 1. Prima di addentrarci nello studio sistematico della Logica e dell'Algebra Booleana, facciamo un esempio introduttivo che ci fa capire fin da subito l'importante valore operativo di questa disciplina.

Problema Si voglia progettare un sistema di allarme per le cinture di sicurezza di un autoveicolo. Le specifiche di progetto sono:

1. si dispone della seguente serie di sensori atti a misurare le condizioni del sistema:
   - un sensore che attesta l'accensione del motore;
   - un sensore, piazzato sul cambio, per accorgersi se esso sia in folle o meno;

- un interruttore posto sotto ciascun sedile anteriore;
- un interruttore connesso con ciascuna cintura, che segnala se essa è allacciata o meno.

2. *Un cicalino deve suonare quando l'accensione è attivata, il cambio è innestato e almeno uno dei due sedili anteriori è occupato senza che la relativa cintura sia allacciata*

Anche se in questo caso il problema è sufficientemente semplice per poter tentare una risoluzione usando metodi di progetto semiempirici, è tuttavia necessario procedere a un'analisi formale delle specifiche assegnate, poiché essa è l'unica praticabile quando si ha a che fare con sistemi più complessi.
Le grandezze delle specifiche di progetto possono essere listate come segue:

suono del cicalino	C
accensione attivata	A
cambio innestato	G
sedile anteriore sinistro occupato	S_L
sedile anteriore destro occupato	S_R
cintura sinistra allacciata	B_L
cintura destra allacciata	B_R

A ciascuna grandezza viene associata una variabile, rappresentativa della grandezza stessa, che assume un valore di verità V o F (*Vero* o *Falso*) a seconda che la condizione espressa sia *vera* o *falsa*; chiamiamo tale variabile *variabile Booleana*. Ciò si può generalizzare a qualsiasi dichiarazione che possa essere classificata come vera o falsa; in tal caso la dichiarazione potrebbe essere costruita con l'apporto di più variabili Booleane opportunamente combinate.
Il metodo che permette di manipolare queste variabili e di assegnare a esse un valore di verità è conosciuto come *calcolo funzionale di verità* e non si limita alla manipolazione di grandezze semplici come quelle appena introdotte.
Si può innanzitutto osservare che per ogni affermazione assertiva esiste una corrispondente formulazione negativa, che viene espressa mediante una barra orizzontale sopra la variabile corrispondente. Ad esempio:

$$\textit{La cintura sinistra non è allacciata} \quad \rightarrow \quad \overline{B_L}$$

Poiché $\overline{B_L}$ è vera quando B_L è falsa e viceversa, essa viene chiamata la *negazione* di B_L, e viene indicata anche con NOT(B_L). Spesso la negazione di una variabile A viene indicata simbolicamente anche con $\sim A$ o con A'.
Si consideri ora la proposizione:

$$\textit{La cintura di sinistra non è allacciata e il sedile anteriore sinistro è occupato} \quad \rightarrow \quad \overline{B_L} \cap S_L$$

Quest'ultimo è un esempio di *funzione composta di verità* o *affermazione composta*, il cui valore di verità può essere determinato a partire dalle proposizioni componenti. Le esatte relazioni tra i valori di verità delle affermazioni componenti e il valore di verità dell'affermazione composta dipende dalla connessione (o connettivo) esistente tra le parti componenti. Nel caso in esame il connettivo e l'AND e indica che la proposizione $\overline{B_L} \cap S_L$ è vera *se e solo se* sono vere entrambe le proposizioni elementari $\overline{B_L}$ e S_L. L'AND è una relazione molto comune tra le proposizioni e viene rappresentata con il simbolo $\cap$.
Vi sono evidentemente solo quattro possibili combinazioni di valori di verità di due proposizioni A e B. Si può pertanto definire completamente la proposizione composta $A \cap B$ riportando i suoi valori di verità in una tabella di quattro righe, come illustrato nella terza colonna della tabella di figura 4.1; tale tabella prende il nome di *tavola di verità* (del connettivo AND, in questo caso).

		AND	OR	XOR	XNOR	NOR	NAND
A	B	$A \cap B$	$A \cup B$	$A \oplus B$	$A \equiv B$	$A \downarrow B$	$A \mid B$
F	F	F	F	F	V	V	V
F	V	F	V	V	F	F	V
V	F	F	V	V	F	F	V
V	V	V	V	F	V	F	F

Figura 4.1: Tavola di verità dei principali connettivi

La tabella definisce anche il connettivo $A \cup B$ (quarta colonna in figura 4.1) che simbolizza una proposizione che è vera quando l'una o l'altra o entrambe le proposizioni elementari sono vere. Un esempio della vita reale potrebbe essere rappresentato dalla frase

Se hai fame o sete ti offro qualcosa al bar

Il relativo connettivo viene chiamato OR; è evidente che in questo caso si va al bar se l'amico è assetato, affamato o entrambi (a maggior ragione). Ma l'uso comune della congiunzione "o" non è sempre in accordo col significato della proposizione $A \cup B$. Si consideri infatti l'affermazione:

Giorgio usa vestiti grigi o azzurri

E' evidente che tale affermazione non significa che Giorgio usi contemporaneamente vestiti grigi e azzurri. Pertanto, nel caso che si sta esaminando il connettivo prende il nome di OR ESCLUSIVO (o XOR, che è la contrazione di *eXclusive OR*) e viene rappresentato con $A \oplus B$. La relativa tavola di verità è riportata nella quinta colonna di figura 4.1. Per sottolineare la differenza con quest'ultimo caso, il connettivo $\cup$ viene per contrasto chiamato OR INCLUSIVO, ma molto spesso, in relazione al suo frequentissimo uso, lo si chiama semplicemente OR.
Prima di passare al calcolo funzionale di verità del problema che si sta esaminando, definiamo anche altri connettivi di uso frequente. Il primo è $A \equiv B$; esso sta a indicare che A e B hanno sempre lo stesso valore di verità, cioè A è vero *se e solamente se* B è vero. Poiché esso è la negazione dello XOR, viene anche chiamato XNOR. La relativa tavola di verità è rappresentata in sesta colonna della figura 4.1.
Gli ultimi due connettivi che introduciamo sono il connettivo NOR e il connettivo NAND, i cui simboli sono rispettivamente $\downarrow$ e $|$; essi rappresentano la negazione di OR e di AND. I valori di verità occupano la settima e l'ottava colonna della tabella di figura 4.1.

Si possono ora rappresentare le specifiche di progetto del sistema in esame con un'equazione funzionale di verità. E' tuttavia opportuno manipolare tali specifiche in termini di proposizioni semplici in modo da eliminare le ambiguità lessicali:

Specifiche - L'allarme suonerà (C) se e solo se valgono contemporaneamente queste tre condizioni:

1. l'accensione è attivata (A)
2. il cambio è innestato (G)
3. - il sedile anteriore sinistro è occupato (S_L) e la cintura sinistra non è allacciata ($\overline{B_L}$) oppure
   - il sedile anteriore destro è occupata (S_R) e la cintura destra non è allacciata ($\overline{B_R}$)

A partire da tali specifiche si può scrivere facilmente l'equazione funzionale di verità. Si ottiene:

$$C \equiv A \cap G \cap [(S_L \cap \overline{B_L}) \cup (S_R \cap \overline{B_R})] \tag{4.1}$$

Si osservi che la rigorosa struttura gerarchica data alle specifiche è utile a evitare ambiguità nella precisa dichiarazione delle stesse, in particolare nei connettivi logici che legano le varie dichiarazioni. E' infatti molto facile incappare in tali ambiguità quando si descrivono specifiche mediante proposizioni dichiarative, usando però un linguaggio colloquiale. Un esempio in tal senso potrebbe essere la frase

L'allarme suona quando la cintura destra non è allacciata

che è una frase di senso comune, ben comprensibile, ma che non rappresenta in modo adeguato la complessità e la struttura delle specifiche. Questa infatti non è una funzione composta di verità, poiché il suo valore di verità non può essere determinato unicamente a partire a partire dal valore di verità delle singole proposizioni componenti. Infatti, anche se ambedue le componenti sono vere, la proposizione composta può non essere vera in quanto il sedile destro potrebbe non essere occupato. In tal caso la parte destra dell'affermazione composta è vera, ma l'allarme potrebbe essere stato attivato dal conducente che non ha allacciato la cintura di sicurezza. Un'affermazione del tipo appena visto non può evidentemente essere manipolata tramite il calcolo funzionale di verità.

4.1.2 I connettivi binari

Nella tabella di figura 4.1 sono stati individuati sei connettivi logici, AND, OR, XOR e XNOR, NOR e NAND, che sono i più interessanti. Poiché per ciascuna coppia di valori delle variabili ci sono due possibilità, abbiamo 4 righe, e quindi $2^4 = 16$ connettivi logici. Vediamoli tutti:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			AND					XOR	OR	NOR	XNOR					NAND	
A	B		$\cap$					$\oplus$	$\cup$	$\downarrow$	$\equiv$		$\subset$		$\supset$	$\mid$	
F	F	F	F	F	F	F	F	F	F	V	V	V	V	V	V	V	V
F	V	F	F	F	F	V	V	V	V	F	F	F	F	V	V	V	V
V	F	F	F	V	V	F	F	V	V	F	F	V	V	F	F	V	V
V	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V

Figura 4.2: Tavola di verità dei $2^4 = 16$ connettivi binari

Tra questi vari connettivi, veste particolare interesse quello corrispondente al numero 13, che si indica con

$$A \supset B$$

che corrisponde alla proposizione composta

se A è vero, allora B è vero if A then B

ed è chiamato *implicazione*. Questo connettivo è simile allo XNOR "*se e solamente se*" introdotto al paragrafo precedente (connettivo 9), eccetto per il fatto che B non è necessariamente falso se A è falso.
Si consideri ad esempio la seguente proposizione:

se piove allora ci sono nuvole in cielo

l'unica possibilità che tale affermazione sia falsa è quella di verificare che in un dato momento piova ma *non* ci siano nuvole in cielo. Supponendo che sia vera $A \supset B$, questa può anche essere espressa nei seguenti modi:

A è condizione sufficiente per B
B è condizione necessaria per A

Applicando tali modi all'esempio precedente rispetto al linguaggio comune possiamo affermare che condizione sufficiente perché in cielo ci siano nuvole è che piova; oppure che condizione necessaria perché piova è che in cielo ci siano nuvole.
Vediamo ora un altro esempio:

$$(2 > x > 1) \supset (x > 0) \tag{4.2}$$

Se $x = 0.5$ la proposizione $x > 0$ è certamente vera, mentre la proposizione $2 > x > 1$ è falsa. Non vi è tuttavia contraddizione con la proposizione 4.2 nel suo complesso, che è vera come è stato definito nella tabella di figura 4.2. Il valore di verità dell'implicazione (che spesso è anche detta if - then) può a prima vista apparire non molto naturale, ma questo concetto è necessario in molti argomenti matematici. E' tuttavia bene puntualizzare che la distinzione tra il connettivo *implicazione* e quello "*se e solamente se*" è estremamente importante. Infatti la validità di quest'ultimo è subordinata alla contemporanea validità dell'implicazione nei due sensi

$$A \equiv B \iff A \supset B \text{ e } A \subset B$$

cioè (4.3)

$$(A \equiv B) \equiv (A \supset B) \cap (A \subset B)$$

La dimostrazione è immediata e si può ottenere manipolando e confrontando le rispettive tavole di verità.

A	B	$A \supset B$	$A \subset B$	$(A \supset B) \cap (A \subset B)$	$(A \equiv B)$
F	F	V	V	V	V
F	V	V	F	F	F
V	F	F	V	F	F
V	V	V	V	V	V

4.1.3 Insiemi minimi di connettivi

La relazione (4.3) ci mostra che alcuni connettivi binari possono essere espressi sulla base di altri connettivi; in questo caso $\equiv$ viene espresso in funzione di $\supset, \subset$ e $\cap$. Si presenta quindi spontanea la domanda di quali siano i connettivi idonei a esprimere tutti gli altri, e quale sia l'insieme minimo tra essi che permette di esprimere tutte le 16 combinazioni di figura 4.2. Si consideri ad esempio l'insieme dei connettivi che contiene solo $\cup$ e $\cap$, cioè OR e AND. I casi in cui sia A che B sono falsi è chiaramente critico; infatti dalla tabella di figura 4.1 vediamo che sia $A \cup B$ e $A \cap B$ sono ambedue falsi; quindi, sulla base di questi due soli connettivi, non è possibile esprimere connettivi che assumono valore vero quando A e B sono falsi. Per poter esprimere anche questi casi, è necessario arricchire l'insieme introducendo anche il connettivo NOT. Ecco allora che l'insieme dei connettivi AND, OR, NOT è un insieme sufficiente per ricostruire tutti gli altri connettivi, come si può verificare dalla tabella di verità di figura 4.3.

$F\,F\,V\,V$ $F\,V\,F\,V$	A B		$F\,F\,V\,V$ $F\,V\,F\,V$	A B	
$F\,F\,F\,F$	$f_0 \equiv A \cap \overline{A}$		$V\,F\,F\,F$	$f_8 \equiv \overline{A \cup B}$	NOR
$F\,F\,F\,V$	$f_1 \equiv A \cap B$	AND	$V\,F\,F\,V$	$f_9 \equiv (A \cup \overline{B}) \cap (\overline{A} \cup B)$	XNOR
$F\,F\,V\,F$	$f_2 \equiv A \cap \overline{B}$			$\equiv (\overline{A} \cap \overline{B}) \cup (A \cap B)$	
$F\,F\,V\,V$	$f_3 \equiv A$		$V\,F\,V\,F$	$f_{10} \equiv \overline{B}$	
$F\,V\,F\,F$	$f_4 \equiv \overline{A} \cap B$		$V\,F\,V\,V$	$f_{11} \equiv A \cup \overline{B}$	$\subset$
$F\,V\,F\,V$	$f_5 \equiv B$		$V\,V\,F\,F$	$f_{12} \equiv \overline{A}$	
$F\,V\,V\,F$	$f_6 \equiv (A \cap \overline{B}) \cup (\overline{A} \cap B)$	XOR	$V\,V\,F\,V$	$f_{13} \equiv \overline{A} \cup B$	$\supset$
	$\equiv (\overline{A} \cup \overline{B}) \cap (A \cup B)$		$V\,V\,V\,F$	$f_{14} \equiv \overline{A \cap B}$	NAND
$F\,V\,V\,V$	$f_7 \equiv A \cup B$	OR	$V\,V\,V\,V$	$f_{15} \equiv A \cup \overline{A}$	

Figura 4.3: I 16 connettivi binari della tabella di figura 4.2 espressi mediante AND, OR, NOT

Per costruire la tabella basta partire da A e B e costruire in modo sistematico tutte le unioni, intersezioni e negazioni tra i vari elementi. P.es. da $A = FFVV$ ricavo $\overline{A} = VVFF$, da $B = FVFV$ ricavo $\overline{B} = VFVF$, e da questi $A \cap B = FFFV$, $A \cap \overline{B} = FFVF$, $A \cup B = VVVF$ ecc. Così facendo si riempie quasi interamente la tabella, cioè quasi tutti i connettivi $f_0 ... f_{15}$ possono essere costruiti usando solo il connettivo $\cup$ oppure solo il connettivo $\cap$ (con l'eventuale negazione). L'unica eccezione è costituita dai connettivi f_6 e f_9, che necessitano l'impiego contemporaneo di $\cup$ e $\cap$. L'espressione risolutiva per f_9 si può ricavare tenendo conto dell'espressione (4.3) e del fatto che $A \supset B = \overline{A} \cup B$ e $A \subset B = A \cup \overline{B}$, e dunque $f_9 \equiv (A \cup \overline{B}) \cap (\overline{A} \cup B)$, mentre la f_6 è la sua negazione.
E' interessante notare che ci sono due modi per realizzare f_6 e f_9, combinando assieme $\cup$ e $\cap$, ma a ben guardare un po' tutti i connettivi possono essere espressi in modi alternativi; p.es. $f_0 \equiv A \cap \overline{A}$, ma anche $f_0 \equiv \overline{A \cup \overline{A}}$, e così via. Tutto ciò deriva dalla circostanza che l'insieme dei connettivi AND, OR, NOT non è il minimo; il connettivo AND può infatti essere espresso in funzione del connettivo OR (e viceversa); a tal riguardo vale l'importante teorema di De Morgan

Teorema 4.1 (De Morgan)**.**

$$\overline{A \cup B} \equiv \overline{A} \cap \overline{B} \qquad \overline{A \cap B} \equiv \overline{A} \cup \overline{B} \tag{4.4}$$

La dimostrazione procede per induzione perfetta, cioè effettuando la verifica direttamente sulle tavole di verità

A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cap \overline{B}$	$A \cup B$	$\overline{A \cup B}$
F	F	V	V	V	F	V
F	V	V	F	F	V	F
V	F	F	V	F	V	F
V	V	F	F	F	V	F

A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cup \overline{B}$	$A \cap B$	$\overline{A \cap B}$
F	F	V	V	V	F	V
F	V	V	F	V	F	V
V	F	F	V	V	F	V
V	V	F	F	F	V	F

(4.5)

Il teorema 4.1 si può generalizzare al caso di più variabili; accade infatti che la negazione di un connettivo di più variabili si ottiene negando ogni variabile e scambiando tra di loro i due operatori $\cup$ e $\cap$. In termini formali si ha

$$\overline{F}(x_1, x_2, \cdots, x_n)[\cup, \cap] = F(\overline{x_1}, \overline{x_2}, \cdots, \overline{x_n})[\cap, \cup] \tag{4.6}$$

I connettivi che legano le variabili non negate si esprimono allora come

$$A \cap B \equiv \overline{\overline{A} \cup \overline{B}} \qquad A \cup B \equiv \overline{\overline{A} \cap \overline{B}} \tag{4.7}$$

Poiché l'AND può essere espresso in funzione di OR e NOT, quest'ultimo insieme è da solo sufficiente per coprire tutti i connettivi di tabella 4.3, che sono riportati nella prima colonna della tabella 4.4. Per farlo bisogna sostituire

tutti i simboli $\cap$ con la prima espressione della (4.7); così facendo si ottiene la seconda colonna della tabella 4.4. Lo stesso ragionamento può essere fatto per l'OR, che può essere espresso in funzione di AND e NOT; per farlo bisogna sostituire tutti i simboli $\cup$ con la seconda espressione della (4.7); così facendo si ottiene la terza colonna della tabella 4.4.

	AND, OR, NOT	OR, NOT	AND, NOT
	$f_0 \equiv A \cap \overline{A}$	$\overline{\overline{A} \cup A}$	$A \cap \overline{A}$
AND	$f_1 \equiv A \cap B$	$\overline{\overline{A} \cup \overline{B}}$	$A \cap B$
	$f_2 \equiv A \cap \overline{B}$	$\overline{\overline{A} \cup B}$	$A \cap \overline{B}$
	$f_3 \equiv A$	A	A
	$f_4 \equiv \overline{A} \cap B$	$\overline{A \cup \overline{B}}$	$\overline{A} \cap B$
	$f_5 \equiv B$	B	B
XOR	$f_6 \equiv (A \cap \overline{B}) \cup (\overline{A} \cap B)$	$\overline{(\overline{A} \cup B)} \cup \overline{(A \cup \overline{B})}$	$\overline{\overline{(A \cap \overline{B})} \cap \overline{(\overline{A} \cap B)}}$
OR	$f_7 \equiv A \cup B$	$A \cup B$	$\overline{\overline{A} \cap \overline{B}}$
NOR	$f_8 \equiv \overline{A \cup B}$	$\overline{A \cup B}$	$\overline{A} \cap \overline{B}$
XNOR	$f_9 \equiv (A \cup \overline{B}) \cap (\overline{A} \cup B)$	$\overline{\overline{(A \cup \overline{B})} \cup \overline{(\overline{A} \cup B)}}$	$\overline{(\overline{A} \cap B)} \cap \overline{(A \cap \overline{B})}$
	$f_{10} \equiv \overline{B}$	$\overline{B}$	$\overline{B}$
$\subset$	$f_{11} \equiv A \cup \overline{B}$	$A \cup \overline{B}$	$\overline{\overline{A} \cap B}$
	$f_{12} \equiv \overline{A}$	$\overline{A}$	$\overline{A}$
$\supset$	$f_{13} \equiv \overline{A} \cup B$	$\overline{A} \cup B$	$\overline{A \cap \overline{B}}$
NAND	$f_{14} \equiv \overline{A \cap B}$	$\overline{A} \cup \overline{B}$	$\overline{A \cap B}$
	$f_{15} \equiv A \cup \overline{A}$	$A \cup \overline{A}$	$\overline{\overline{A} \cap A}$

Figura 4.4: I 16 connettivi binari espressi rispettivamente mediante AND, OR, NOT, mediante OR, NOT e mediante AND, NOT

Osserviamo ora che l'insieme minimo che consente di rappresentare tutti i connettivi può essere ulteriormente ridotto al solo connettivo NOR, oppure al solo connettivo NAND; per farlo è sufficiente esprimere AND, OR e NOT in funzione del solo NOR oppure del solo NAND. Per questo motivo i connettivi NOR e NAND sono chiamati *universali*. Ecco come procedere:

$$
\begin{aligned}
\overline{A} &\equiv \overline{A \cup A} \equiv A \downarrow A & \overline{A} &\equiv \overline{A \cap A} \equiv A|A \\
A \cup B &\equiv \overline{\overline{A \cup B}} \equiv \overline{A \downarrow B} & A \cup B &\equiv \overline{\overline{A \cup B}} \equiv \overline{\overline{A} \cap \overline{B}} \equiv \overline{A}|\overline{B} \\
&\equiv (A \downarrow B) \downarrow (A \downarrow B) & &\equiv (A|A)|(B|B) \\
A \cap B &\equiv \overline{\overline{A \cap B}} \equiv \overline{\overline{A} \cup \overline{B}} \equiv \overline{A} \downarrow \overline{B} & A \cap B &\equiv \overline{\overline{A \cap B}} \equiv \overline{A|B} \\
&\equiv (A \downarrow A) \downarrow (B \downarrow B) & &\equiv (A|B)|(A|B)
\end{aligned}
\tag{4.8}
$$

Si osservi che invece *non* è possibile ottenere il NOT usando solo AND o solo OR.
Come vedremo nel seguito, il fatto che AND-NOT e OR-NOT siano insiemi minimi grazie ai quali possiamo rappresentare tutte le 16 possibili combinazioni di tabella 4.3 e che NOR e NAND siano connettivi universali, avrà un'importanza fondamentale in ambito tecnico-applicativo.

4.2 Algebra Booleana

4.2.1 Impostazione assiomatica

La formalizzazione e la soluzione di problemi del mondo reale, quale quello appena analizzato riguardante le cinture di sicurezza, richiede di poter usare le tavole di verità associate alle variabili binarie T e F, ma soprattutto di poter manipolare in modo rigoroso tali variabili e i rispettivi connettivi binari. Inoltre, potrebbe anche accadere che la formula 4.1 abbia delle espressioni equivalenti, che sono magari più semplici o più utili ai fini applicativi; bisogna allora essere in grado di esprimere in modo rigoroso queste formulazioni equivalenti. E' dunque necessario dotarsi di una tecnica affidabile di manipolazione delle espressioni del calcolo funzionale di verità.

In matematica, la disciplina che studia i simboli matematici e le regole per la loro manipolazione formale si chiama *Algebra*; l'Algebra che studia i valori logici *vero* e *falso* si chiama *Algebra Booleana*, e come ricordato precedentemente essa venne introdotta nel 1854 da George Boole, con l'opera *An Investigation of the Laws of Thought*. Vediamone una definizione assiomatica.

Definizione 4.1. *Un'Algebra Booleana è un insieme $\mathcal{B}$ caratterizzato da:*

A0 - Leggi di composizione *Ci sono due leggi di composizione interna di-arie, denominate rispettivamente AND e OR, e una legge di composizione interna unaria, denominata NOT; tali leggi (od "operatori Booleani") sono definite come segue:*

Operatore	Nome	Simboli usati	che soddisfa
AND(x, y)	congiunzione	$x \wedge y \quad x \cdot y$	$x \cdot y = 1$ se $x = 1, y = 1$
			$x \cdot y = 0$ altrimenti
OR(x, y)	disgiunzione	$x \vee y \quad x + y$	$x + y = 0$ se $x = 0, y = 0$
			$x + y = 1$ altrimenti
NOT(x)	negazione	$\neg x \quad \overline{x}$	$\overline{x} = 0$ se $x = 1$ e $\overline{x} = 1$ se $x = 0$

Inoltre, per gli elementi dell'insieme $\mathcal{B}$ valgono i seguenti assiomi:

A1 - Esistenza Esistono due elementi $x, y \in \mathcal{B}$, tali che $x \neq y$

A2 - Chiusura Se $x, y \in \mathcal{B}$, allora $x \cdot y \in \mathcal{B}$, $x + y \in \mathcal{B}$, $\overline{x} \in \mathcal{B}$, $\overline{y} \in \mathcal{B}$

A3 - Elemento neutro Esiste un elemento neutro per la disgiunzione, indicato con 0, tale che $x + 0 = x$

Esiste un elemento neutro per la congiunzione, indicato con 1, tale che $x \cdot 1 = x$

A4 - Commutatività $\forall x, y \in \mathcal{B}$ valgono le seguenti relazioni $x + y = y + x$

$$x \cdot y = y \cdot x$$

A5 - Associatività $\forall x, y, z \in \mathcal{B}$ valgono le seguenti relazioni $x + (y + z) = (x + y) + z$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

A6 - Distributività $\forall x, y, z \in \mathcal{B}$ valgono le seguenti relazioni $x + (y \cdot z) = (x + y) \cdot (x + z)$;

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

A7 -Complementarità $\forall x \in \mathcal{B}$ vale $x \cdot \overline{x} = 0$ e $x + \overline{x} = 1$

Il più semplice insieme $\mathcal{B}$ che soddisfa i postulati sopra è rappresentato dalla cosiddetta *Algebra Booleana a due elementi*, o *Algebra Binaria*, basata sugli elementi 0 e 1 e caratterizzata dalle seguenti tre leggi di composizione

$$\begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \qquad \begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array} \qquad \begin{array}{c|cc} A & 0 & 1 \\ \hline \overline{A} & 1 & 0 \end{array} \tag{4.9}$$

L'Algebra Booleana a due elementi è quella che useremo per la progettazione delle reti logiche; i suoi due elementi 1 e 0 corrispondono alle costanti logiche *vero* e *falso* (V e F) prima introdotte, e le tre leggi di composizione interna sono rispettivamente AND, OR e NOT e corrispondono ai connettivi precedentemente introdotti nelle tavole di verità. In questo contesto la simbologia che si usa è però diversa; in particolare si osservi che i simboli $+$ e $\cdot$ nulla hanno a che vedere con gli stessi simboli usati nel contesto dell'aritmetica.
Un altro importante esempio di Algebra Booleana è quello derivante dalla teoria degli insiemi; si consideri un insieme S e il corrispondente insieme delle parti $\mathcal{P}(S)$, costituito dall'insieme di tutti i sottinsiemi costruiti con elementi di S. Allora $\mathcal{P}(S)$ è un Algebra Booleana, nella quale sono definite tre leggi di composizione interna date dall'intersezione $\cap$, dall'unione $\cup$ e dalla complementazione tra insiemi. L'elemento neutro per la disgiunzione, o 0 dell'insieme, è costituito dall'insieme vuoto, mentre l'elemento neutro per la congiunzione, o 1 dell'insieme, è l'insieme S.
Disponendo degli operatori Booleani e agendo sugli elementi di $\mathcal{B}$, potremmo ricavare infinite relazioni notevoli, dette *leggi Booleane* o *teoremi*, alcune delle quali sono già state viste precedentemente con le tavole di verità; un esempio potrebbe essere la relazione (4.4) che esprime i teoremi di De Morgan. Una legge Booleana è dunque un'identità tra due termini Booleani, costruiti su un insieme di variabili e sulle costanti 0 e 1 a partire dagli operatori $\wedge, \vee, \neg$. Per verificare una legge è sufficiente procedere con la cosiddetta *induzione perfetta*, che consiste nella sostituzione di tutti i possibili valori per le variabili, verificando che l'uguaglianza sia sempre soddisfatta.

Le leggi Booleane sono ovviamente infinite, ma poiché l'Algebra Booleana è un insieme *finitamente assiomatizzabile*, è possibile descrivere compiutamente l'Algebra Booleana usando solamente un insieme *finito* di leggi base, che sono gli *assiomi* precedentemente descritti. Ciò significa che partendo dagli assiomi si possono poi ricavare tutte le leggi (o teoremi) dell'Algebra Booleana, combinando gli assiomi e i teoremi che ne derivano in tutti i modi possibili.
Gli assiomi non sono soggetti a verifica, nel senso che sono verità primitive che costituiscono il punto di partenza di una teoria matematica. Essi devono inoltre essere non contradditori tra loro e possibilmente indipendenti. Quest'ultima caratteristica è in qualche modo legato al seguente problema: assegnato un insieme di assiomi, esso è il minimo possibile o ne esiste uno più piccolo? Per esempio, nel nostro caso si può osservare che l'operatore disgiunzione $\cdot$, usato come operatore base nel quadro assiomatico appena analizzato, può essere definito in funzione degli altri due operatori $+$ e $\neg$, come già verificato nel teorema di De Morgan 4.1; ecco allora che potremmo rimuoverlo, pervenendo a un quadro assiomatico più sintetico. Il problema fu affrontato da Edward V. Huntington, che nel 1933 riuscì a formulare un insieme minimo di assiomi tra loro indipendenti, che partono dai soli operatori $+$ e $\neg$ e che prevedono, oltre alla commutatività e all'associatività, anche la cosiddetta equazione di Huntington

$$\overline{(\overline{x} + y)} + \overline{(\overline{x} + \overline{y})} = x \tag{4.10}$$

4.2.2 Teoremi principali dell'Algebra Booleana

A partire dagli assiomi sopra esposti, si potrebbero ricavare tutti i possibili teoremi dell'Algebra Booleana; tra questi ce ne sono alcuni molto semplici e utili nella manipolazione delle formule; li elenchiamo di seguito

	base (b)	duale (d)
T1 - Idempotenza	$x + x = x$	$x \cdot x = x$
T2 - Nullifico	$x + 1 = 1$	$x \cdot 0 = 0$
T3 - Doppia negazione	$\overline{\overline{x}} = x$	
T4 - Assorbimento 1	$x + x \cdot y = x$	$x \cdot (x + y) = x$
T5 - Assorbimento 2	$x + \overline{x} \cdot y = x + y$	$x \cdot (\overline{x} + y) = x \cdot y$
T6 - Assorbimento 3	$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$	$(x+y) \cdot (\overline{x}+z) \cdot (y+z) = (x+y) \cdot (\overline{x}+z)$
T7 - De Morgan	$\overline{x + y} = \overline{x} \cdot \overline{y}$	$\overline{x \cdot y} = \overline{x} + \overline{y}$
T8 -	$x \cdot (x + y + z) = x$	$x + (x \cdot y \cdot z) = x$
T9 -	$(x + y) \cdot (\overline{x} + y) = y$	$(x \cdot y) + (\overline{x} \cdot y) = y$
T10 -	$(x + y) \cdot (\overline{x} + z) = x \cdot z + \overline{x} \cdot y$	$(x \cdot y) + (\overline{x} \cdot z) = (x + z) \cdot (\overline{x} + y)$

La dimostrazione di uno qualunque di questi teoremi può essere fatta per induzione perfetta, che è la via più lunga e tediosa, oppure sfruttando gli assiomi e/o i teoremi già dimostrati. Vediamo alcune dimostrazioni tralasciando il segno "·", per non appesantire la notazione. T1-T3 sono banali, poichè basta fare una verifica diretta tramite induzione perfetta. Vediamo la dimostrazione di T4, T5 e T6, mettendo tra parentesi, subito dopo il segno di uguaglianza, l'assioma o il teorema usato.

T4 - Assorbimento 1

base $x + xy \overset{(A3)}{=} x \cdot 1 + xy \overset{(A6)}{=} x(1 + y) \overset{(T2)}{=} x$

duale $x(x + y) \overset{(A6)}{=} xx + xy \overset{(T1)}{=} x + xy \overset{(T4.b)}{=} x$

T5 - Assorbimento 2

base $x + \overline{x}y \overset{(T2)}{=} x(1 + y) + \overline{x}y \overset{(A6)}{=} x + xy + \overline{x}y \overset{(A6)}{=} x + (x + \overline{x})y \overset{(A7)}{=} x + y$

duale $x(\overline{x} + y) \overset{(A6)}{=} x\overline{x} + xy \overset{(A7)}{=} xy$

T6 - Assorbimento 3

base $xy + \overline{x}z + yz \overset{(A7)}{=} xy + \overline{x}z + yz(x + \overline{x}) \overset{(A6)}{=} xy + \overline{x}z + yzx + yz\overline{x} \overset{(A6)}{=}$
$xy(z + 1) + \overline{x}z(y + 1) \overset{(T2)}{=} xy + \overline{x}z$

duale $(x + y)(\overline{x} + z)(y + z) \overset{(A7)}{=} (x + y)(\overline{x} + z)(y + z + x\overline{x}) \overset{(A6)}{=}$
$(x + y)(\overline{x} + z)(y + z + x)(y + z + \overline{x}) \overset{(A4)}{=}$
$(x + y)(x + y + z)(\overline{x} + z)(\overline{x} + z + y) \overset{(T4)}{=} (x + y)(\overline{x} + z)$

La verifica del teorema T7 di De Morgan, basata l'induzione perfetta, è già stata fatta nella dimostrazione 4.5 del teorema 4.1; la ripetiamo qua per comodità del lettore, ricordando la fondamentale importanza di questo teorema per le applicazioni e per la semplificazione delle espressioni complesse

x	y	$x + y$	$\overline{x + y}$	$\overline{x}$	$\overline{y}$	$\overline{x} \cdot \overline{y}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Lasciamo infine i teoremi T8-T9 e T10 per esercizio al lettore.

4.2.3 Principio di dualità

Se osserviamo l'elenco dei teoremi, possiamo notare che le proposizioni della colonna di destra possono essere ottenute dalla colonna centrale semplicemente scambiando "0" con "1" e "+" con "·" e viceversa. Questa è una manifestazione del cosiddetto *Principio di dualità*. Per comprenderlo riprendiamo la tabella di figura 4.1, limitatamente ai valori di AND e OR, e facciamo lo scambio "0" con "1" e "+" con "·"; si ottiene

x	y	$x+y$	$x\cdot y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

x	y	$x\cdot y$	$x+y$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Figura 4.5: Principio di dualità: scambiando "0" con "1" e "+" con "·" la tavola di verità continua a valere

Si può osservare che la tavola di verità che ne deriva conserva la propria validità, sia pur con una permutazione delle righe, del tutto ininfluente. Il Principio di dualità consente allora di trasformare in modo duale le espressioni Booleane, realizzando nuove espressioni che continuano a valere. Si osservi tuttavia che i valori delle espressioni così ottenute sono in generale diversi da quelle di partenza; per charire meglio il concetto riprendiamo p.es. il teorema T10; l'espressione Booleana $(x+y)\cdot(\overline{x}+z)$ ha come duale la $x\cdot y+\overline{x}\cdot z$.

1	2	3	4	5
x	y	z	$(x+y)\cdot(\overline{x}+z)$	$x\cdot y+\overline{x}\cdot z$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

(a)

6	7	8	9
x	y	z	$x\cdot y+\overline{x}\cdot z$
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

(b)

Figura 4.6: Principio di dualità applicato all'espressione Booleana $(x+y)\cdot(\overline{x}+z)$

Nella tabella di figura 4.6a sono riportati, in colonna 4 e 5, i valori di queste due espressioni duali, e come si vede sono diversi. Per verificare la validità del Principio di dualità dobbiamo ora sostituire "0" con "1" e "+" con "·" nelle colonne $1, 2, 3$ e nella colonna 4, che riporta i valori dell'espressione; così facendo otteniamo rispettivamente le colonne $6, 7, 8$ e 9. Si può verificare che a ciascuna terna delle colonne duali $6, 7, 8$ corrisponde un valore duale di colonna 9 che è esattamente il valore attribuito dall'espressione di colonna 5 alla stessa terna letta nelle colonne primitive $2, 3, 4$. Per esempio, a 011 delle colonne $6, 7, 8$ corrisponde 1 in colonna 9; questo è lo stesso valore attribuito dalla colonna 5 alla terna 011 delle colonne primitive $1, 2, 3$.

4.3 Variabili, funzioni Booleane e porte logiche

Se osserviamo le colonne 4 o 5 della tabella di figura 4.6, vediamo che a ogni terna di valori per le variabili Booleane x, y, z viene associato un valore per l'espressione corrispondente. In Matematica un'associazione di questo genere si chiama *funzione*; in generale una funzione è una relazione tra due insiemi, chiamati rispettivamente *dominio* e *codominio* della funzione, che associa a ogni elemento del dominio uno e un solo elemento del codominio. Se chiamiamo X e Y i due insiemi, una generica funzione si indica con il simbolo $f : X \to Y$. Nel nostro caso l'insieme X corrisponde all'insieme di tutte e sole le possibili 2^3 terne binarie, che possiamo rappresentare col simbolo $X = \mathbf{2^3}$; [1] Y è invece l'insieme $\{0, 1\}$, che rappresentiamo analogamente col simbolo $Y = \mathbf{2}$. Ecco allora che la funzione rappresentata dalla colonna 4 della tabella di figura 4.6 è una funzione del tipo $f : \mathbf{2^3} \to \mathbf{2}$. Se dunque $x_1, x_2, \cdots, x_n$ sono n variabili Booleane (o *binarie*) che possono assumere l'uno o l'altro dei due valori 0 e 1, si indica con

$$f(x_1, x_2, \cdots, x_n)$$

una generica *funzione Booleana* del tipo

$$f : \mathbf{2^n} \to \mathbf{2}$$

che a ogni valore dell'n-pla $x_1, x_2, \cdots, x_n$ associa un valore dell'insieme $\{0, 1\}$. La tabella di verità di una simile funzione è rappresentata in figura 4.7.

x_1	x_2	$\cdots$	x_{n-1}	x_n	$f(x_1, x_2, \cdots, x_n)$
0	0	$\cdots$	0	0	$f(0, 0, \cdots, 0, 0)$
0	0	$\cdots$	0	1	$f(0, 0, \cdots, 0, 1)$
		$\vdots$			$\vdots$
1	1	$\cdots$	1	0	$f(1, 1, \cdots, 1, 0)$
1	1	$\cdots$	1	1	$f(1, 1, \cdots, 1, 1)$

Figura 4.7: Tabella di verità di una generica funzione Booleana n-aria

Si osservi che il numero di n-ple binarie dell'insieme $\mathbf{2^n}$ cresce in modo esponenziale; diventa dunque inagevole rappresentare le funzioni Booleane mediante tavole di verità simili a quella di figura 4.6 o 4.7, se non che per valori molto limitati di n.
Poiché l'insieme $\mathbf{2^n}$ contiene 2^n n-ple binarie, e a ciascuna di esse si associa un valore della funzione, il numero totale di funzioni Booleane n-arie che si possono realizzare risulta essere

$$2^{2^n}$$

Risulta fondamentale, per tutto quanto faremo nel seguito, studiare in particolare le funzioni Booleane per $n = 1$ (a una variabile, o *unarie*) e $n = 2$ (a due variabili, o *di-arie*).

4.3.1 Funzioni a una variabile

Con $n = 1$ ci sono in tutto $2^{2^1} = 4$ funzioni unarie del tipo $y = f(x)$. Passiamole in rassegna:

[1] si osservi che l'uso del grassetto sta proprio a indicare che $\mathbf{2^3}$ non è il numero $2^3 = 8$, ma un simbolo, cioè il simbolo dell'insieme che contiene tutti e sole le 2^3 terne binarie

	f_0	f_1	f_2	f_3
x	$\mathbf{0}$	x	$\bar{x}$	$\mathbf{1}$
0	0	0	1	1
1	0	1	0	1

	$\mathbf{0}$
x	z
0	0
1	0

	$\mathbf{1}$
x	z
0	1
1	1

	Buffer
x	z
0	0
1	1

	NOT
x	z
0	1
1	0

Figura 4.8: Tutte le possibili $2^{2^1} = 4$ funzioni Booleane a 1 variable

La prima funzione fornisce sempre 0 in uscita, qualunque sia l'ingresso; è dunque la *funzione nulla*, che denotiamo con $\mathbf{0}$. Un discorso analogo vale per la seconda funzione, che fornisce sempre 1 in uscita (*funzione unitaria*), e che denotiamo con $\mathbf{1}$. La terza funzione replica il valore di x in uscita, cioè $y = x$, e si tratta dunque della *funzione identità*. Poiché la logica Booleana è connessa, come già anticipato, all'ambito circuitale, per la rappresentazione della funzione identità si usa il termine *Buffer*, e le si attribuisce lo schema circuitale dato da un triangolo; esso corrisponde a un dispositivo che fornisce sull'uscita a destra esattamente cò che si presenta in ingresso a sinistra. La quarta funzione è la più importante dal punto di vista della logica Booleana, poiché è uno dei connettivi base già incontrati, cioè NOT. Esso fornisce in uscita la negazione di quanto sta all'ingresso; si può dunque scrivere $y = \bar{x}$. Per la sua rappresentazione circuitale si usa il triangolo di prima concatenato con un piccolo circoletto, che in tutta la circuiteria logica ha sempre il significato di una *negazione* o *complementazione*.

4.3.2 Funzioni a due variabili

Con $n = 2$ ci sono in tutto $2^{2^2} = 16$ funzioni Booleane di-arie del tipo $z = f(x, y)$. Le abbiamo già implicitamente incontrate in figura 4.2 parlando di connettivi binari; le riportiamo di seguito usando il nuovo linguaggio dell'Algebra Booleana:

		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
			AND					XOR	OR	NOR	XNOR					NAND	
x	y	$\mathbf{0}$	$\cdot$					$\oplus$	$+$	$\downarrow$	$\odot$					$\mid$	$\mathbf{1}$
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figura 4.9: Tutte le possibili $2^{2^2} = 16$ funzioni Booleane a due variabili

Tra tutte queste funzioni sappiamo che alcune sono particolarmente significative, precisamente le sei già evidenziate in figura 4.1, che sono rispettivamente AND, OR, XOR, NAND, NOR e XNOR; nella tabella seguente esse sono espresse, assieme alle rimanenti, mediante i simboli $\cdot$, $+$, $\oplus$, $\mid$, $\downarrow$ e $\odot$, che si usano tradizionalmente nell'ambito dell'Algebra Booleana; si noti che $+$, $\cdot$ e $\odot$ stanno rispettivamente per $\cup$, $\cap$ e $\equiv$, visti nella precedente tabella 4.1 parlando di connettivi.

$f_0 = \mathbf{0}$		$f_8 = x \downarrow y$	NOR
$f_1 = x \cdot y$	AND	$f_9 = x \odot y$	XNOR
$f_2 = x \cdot \overline{y}$		$f_{10} = \overline{y}$	
$f_3 = x$		$f_{11} = x + \overline{y}$	
$f_4 = \overline{x} \cdot y$		$f_{12} = \overline{x}$	
$f_5 = y$		$f_{13} = \overline{x} + y$	
$f_6 = x \oplus y$	XOR	$f_{14} = x \mid y$	NAND
$f_7 = x + y$	OR	$f_{15} = \mathbf{1}$	

Figura 4.10: Le 16 funzioni Booleane della figura 4.9 espresse mediante $+$, $\cdot$, $\oplus$, $\odot$, $\downarrow$, $\mid$ e complementazione

Data l'estrema importanza, in ambito circuitale, delle funzioni AND, OR, XOR, NAND, NOR e XNOR, dette anche *operatori Booleani*, le analizziamo singolarmente facendo riferimento alle tabelle di figura 4.15. Nella stessa figura sono evidenziati anche i simboli schematici, detti *Porte Logiche*, che vengono associati alle suddette funzioni nella descrizione dei circuiti logici.

AND E' detta anche *prodotto logico*. La porta AND restituisce 0 in uscita se anche uno solo dei due valori d'ingresso è pari a 0; restituisce 1 solo quando entrambi gli ingressi sono a 1. Il simbolo circuitale è

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR E' detta anche *somma logica*. La porta OR restituisce 1 in uscita se almeno uno dei due valori d'ingresso è pari a 1; restituisce 0 solo quando entrambi gli ingressi sono a 0. Il simbolo circuitale è

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR E' l'OR esclusivo. La porta XOR restituisce 1 in uscita se o uno o l'altro dei due valori d'ingresso è pari a 1, ma non entrambi; restituisce 0 quando entrambi gli ingressi sono a 0 o a 1. Il simbolo circuitale è

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

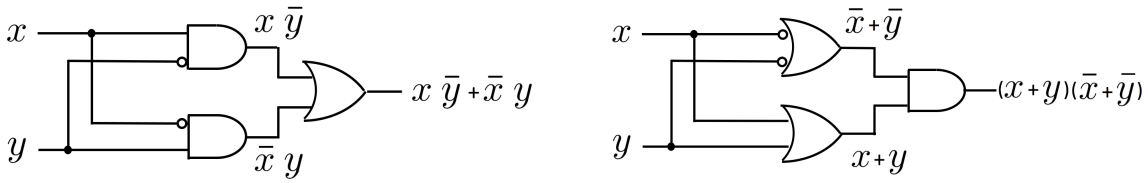
Si osservi che una porta XOR è in grado di realizzare la somma binaria modulo 2 e che costituisce un *rilevatore di differenza* tra i due valori d'ingresso. Dalla figura 4.3 possiamo osservare che la funzione f_6 corrisponde a due espressioni; la prima è

$$x \oplus y = x\overline{y} + \overline{x}y \tag{4.11}$$

che corrisponde a un OR di due AND. Mediante manipolazione algebrica si ottiene anche la seconda espressione

$$x \oplus y = x\overline{y} + \overline{x}y = x\overline{y} + \overline{x}y + x\overline{x} + y\overline{y} = (x + y) \cdot (\overline{x} + \overline{y}) \tag{4.12}$$

che corrisponde a un AND di due OR. Si possono pertanto realizzare due circuiti del tutto equivalenti, visibili in figura 4.11



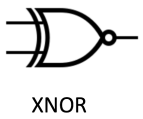
(a) Realizzazione basata sull'equazione 4.11

(b) Realizzazione basata sull'equazione 4.12

Figura 4.11: La funzione XOR realizzata mediante porte AND e OR

XNOR E' la negazione della porta XOR. La porta XNOR restituisce 1 in uscita solo quando entrambe le variabili d'ingresso hanno lo stesso valore, o entrambe a 0 oppure entrambe a 1. Il simbolo circuitale si ottiene concatenando la porta XOR con la porta NOT, rappresentata dal circoletto

x	y	$x \odot y$
0	0	1
0	1	0
1	0	0
1	1	1

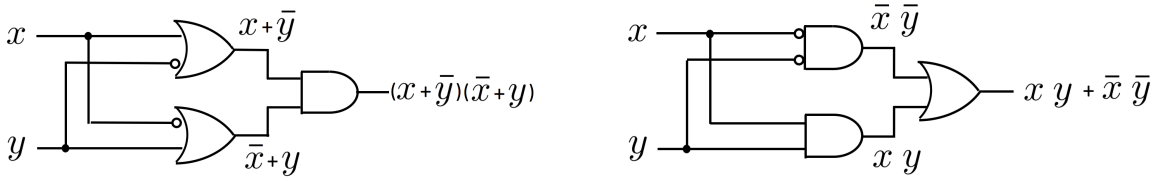


Si osservi che una porta XNOR costituisce un *rilevatore di identità* tra i due valori d'ingresso. Per trovare una rappresentazione circuitale mediante porte AND e OR bisogna partire dal fatto che si tratta di uno XOR negato, traendone le conseguenze sempre usando il teorema T7 di De Morgan

$$x \odot y = \overline{x \oplus y} = \overline{\overline{x}y + x\overline{y}} = \overline{(\overline{x}y)} \cdot \overline{(x\overline{y})} = (x + \overline{y}) \cdot (\overline{x} + y) = \tag{4.13}$$

$$= x\overline{x} + xy + \overline{y}\overline{x} + \overline{y}y = xy + \overline{x}\overline{y} \tag{4.14}$$

Dalla prima riga otteniamo $(x + \overline{y}) \cdot (\overline{x} + y)$, e quindi l'AND di due OR, che corrisponde alla realizzazione circuitale di figura 4.12a; dalla seconda otteniamo invece $xy + \overline{x}\overline{y}$, cioè l'OR di due AND, che porta alla realizzazione circuitale di figura 4.12b



(a) Realizzazione basata sull'equazione 4.13

(b) Realizzazione basata sull'equazione 4.14

Figura 4.12: La funzione XNOR realizzata mediante porte AND e OR

NAND E' la negazione dell'AND. La porta NAND restituisce 1 in uscita se almeno uno dei due valori d'ingresso è pari a 0; restituisce 0 solo quando entrambi gli ingressi sono a 1. Il simbolo circuitale si ottiene concatenando

la porta AND con la porta NOT, rappresentata dal circoletto

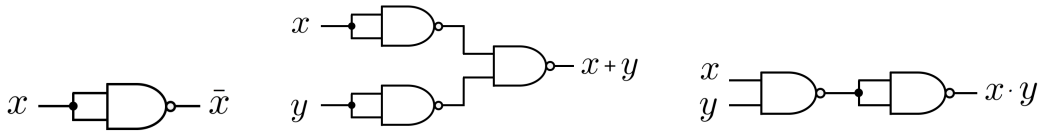
x	y	$x \mid y$
0	0	1
0	1	1
1	0	1
1	1	0

Nella tabella 4.4 avevamo anticipato che AND e NOT, oppure OR e NOT, sono sufficienti per esprimere una qualunque tra le 16 possibili funzioni; inoltre dalle equazioni (4.8) abbiamo visto che tramite una porta NAND si possono realizzare l'AND, l'OR e il NOT; ciò significa che la porta NAND è un operatore *universale*, nel senso che da sola consente di costruire una qualunque tra le 16 possibili funzioni. Per declinare le relazioni (4.8) nei termini del linguaggio dell'Algebra Booleana è sufficiente usare gli assiomi e i teoremi che abbiamo visto poc'anzi

$$\overline{x} = \overline{x \cdot x} = x|x \qquad x + y = \overline{\overline{x} \cdot \overline{y}} = \overline{x}|\overline{y} \qquad x \cdot y = \overline{\overline{x \cdot y}} = \overline{x|y}$$

NOT OR AND

Le figure 4.13a, 4.13b e 4.13c mostrano la realizzazione circuitale corrispondente.



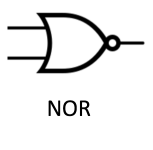
(a) Funzione NOT (b) Funzione OR (c) Funzione AND

Figura 4.13: Funzioni NOT, OR e AND realizzate mediante una porta universale NAND

La giustificazione del NOT è evidente: poichè nel circuito sono ammesse le sole due configurazioni d'ingresso in cui $x = y = 1$ oppure $x = y = 0$, per il primo caso si ottiene l'uscita a 1, che complementata va a 0, realizzando la negazione dell'ingresso; nel secondo caso si ha l'uscita a 0, che complementata va a 1, realizzando nuovamente la negazione dell'ingresso. L'OR si giustifica mediante il teorema T7 di De Morgan; ecco allora che prima di entrare nel NAND bisogna complementare le variabili con un NOT realizzato tramite due porte NAND. Per quanto riguarda la costruzione dell'AND è evidente che esso si realizza aggiungendo un NOT alla porta NAND.

NOR E' la negazione dell'OR. La porta NOR restituisce 0 in uscita se anche uno solo dei due valori d'ingresso è pari a 1; restituisce 1 solo quando entrambi gli ingressi sono a 0. Il simbolo circuitale si ottiene concatenando la porta OR con la porta NOT, rappresentata dal circoletto

x	y	$x \downarrow y$
0	0	1
0	1	0
1	0	0
1	1	0

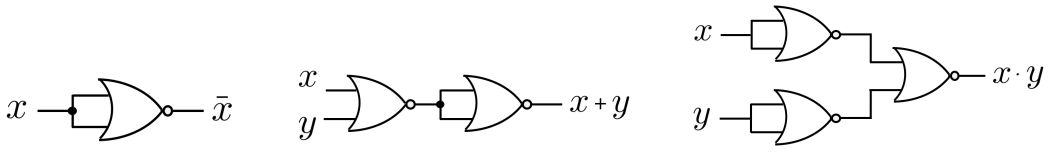


Ricordiamo che anche l'operatore NOR è un operatore *universale*, che consente di costruire una qualunque tra le 16 possibili funzioni; ciò deriva dalla circostanza che, similmente al caso della porta NAND, col NOR si possono costruire le porte NOT, OR e AND mediante l'uso degli assiomi e dei teoremi prima visti

$$\overline{x} = \overline{x + x} = x \downarrow x \qquad x + y = \overline{\overline{x + y}} = \overline{x \downarrow y} \qquad x \cdot y = \overline{\overline{x} + \overline{y}} = \overline{x} \downarrow \overline{y}$$

NOT OR AND

Le figure 4.14a, 4.14b e 4.14c mostrano la realizzazione circuitale corrispondente.



(a) Funzione NOT (b) Funzione OR (c) Funzione AND

Figura 4.14: Funzioni NOT, OR e AND realizzate mediante una porta universale NOR

La giustificazione del NOT è identica a prima: poichè nel circuito sono ammesse le sole due configurazioni d'ingresso in cui $x = y = 1$ oppure $x = y = 0$, nel primo caso l'uscita è a 1 e complementata va a 0, realizzando la negazione dell'ingresso; nel secondo caso l'uscita è a 0 e complementata va a 1, realizzando nuovamente la negazione dell'ingresso. Per quanto riguarda la costruzione dell'OR è evidente che esso si realizza aggiungendo un NOT alla porta NOR. L'AND si giustifica invece mediante il teorema T7 di De Morgan; ecco allora che prima di entrare nel NOR bisogna complementare le variabili con un NOT realizzato tramite due porte NOR. Si osservi che questa relazione è la duale di quella usata per il NAND.

AND

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

NAND

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

NOR

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

x	y	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Figura 4.15: Le 6 funzioni di-arie più importanti con i rispettivi simboli circuitali

Finora abbiamo trattato solo porte associate a funzioni a due variabili; la generalizzazione a n variabili ha pieno senso solo per le porte AND e OR; trattiamo per esteso il caso con $n = 3$, poiché gli altri sono la banale generalizzazione di questo. Le tavole di verità delle funzioni AND e OR a 3 variabili sono riportate nella successiva figura 4.16; la funzione AND vale 1 solo quando tutte le variabili in ingresso hanno valore 1; la funzione OR vale 0 solo quando tutte le variabili in ingresso hanno valore 0.

x	y	z	AND
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a) La funzione AND a 3 variabili

x	y	z	OR
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

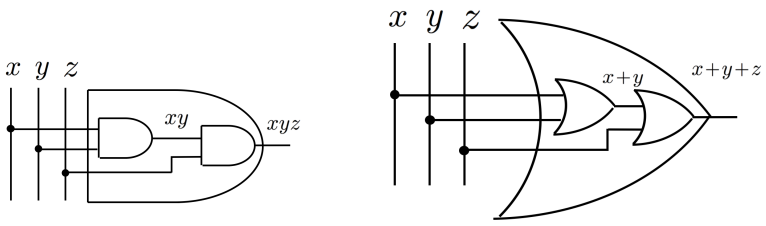
(b) La funzione OR a 3 variabili

Figura 4.16: Le funzioni AND e OR a 3 variabili

Si osservi che per l'associatività A5 di prodotto e somma logiche si ha

$$x + y + z = (x + y) + z \qquad\qquad x \cdot y \cdot z = (x \cdot y) \cdot z$$

e dunque una porta AND o una porta OR a tre variabili si realizza usando due porte a due variabili, collegate come in figura 4.17. L'associatività A5 di prodotto e somma logiche costituisce anche una giustificazione teorica alla circostanza di poter gestire con connettivi binari quali AND e OR a due variabili funzioni aventi un numero qualunque di variabili.



(a) Porta AND a 3 variabili

(b) Porta OR a 3 variabili

Figura 4.17: Porte AND e OR a 3 variabili

4.3.3 Realizzazione circuitale delle porte logiche

Le funzioni logiche di base, a 1 e 2 variabili, possono essere facilmente realizzate impiegando i dispositivi che abbiamo introdotto nel capitolo 3, cioè i tubi termoionici, i diodi, i transistor e i C-MOS. Lasciando perdere i tubi a vuoto, la cui tecnologia è ampiamente superata, la realizzazione delle porte logiche si è sviluppata nell'arco temporale perseguendo obiettivi di minimazione della potenza dissipata, dei tempi di risposta e dei costi, ottenendo congiuntamente un aumento progressivo della densità di integrazione sui relativi circuiti integrati. Il risultato è stato la costituzione delle cosiddette *famiglie logiche*, che a seconda del periodo in cui sono state introdotte e dello sviluppo tecnologico corrente hanno privilegiato l'uso di uno o dell'altro dei vari dispositivi a semiconduttore per la realizzazione delle funzioni di base. Anche se in pratica sono molte di più, le principali famiglie logiche sono le seguenti (in ordine cronologico):

Resistor-Transistor Logic (RTL) è una classe di porte logiche costruite usando resistenze nella rete d'ingresso e transistori bipolari a giunzione (BJT) come dispositivi di commutazione. La RTL è stata la prima classe di

circuito logico digitale basata sull'impiego dei transistor. Le prime porte RTL furono costruite con elementi discreti, ma nel 1961 divenne la prima famiglia logica a esser realizzata su un circuito integrato monolitico. Circuiti integrati RTL sono stati utilizzati nel computer *Apollo Guidance*, il cui progetto risale al 1961 con un primo volo fatto nel 1966.

Diode-Transistor Logic (DTL) è la classe di di porte logiche che precede la grande famiglia TTL. Si chiama così perché la funzione logica viene eseguita da una rete di diodi, mentre la funzione di inversione-amplificazione viene eseguita da un transistore.

Transistor-Transistor Logic (TTL) è senza dubbio la classe più nota e famosa, poiché ha avuto un larghissimo impiego. Fa uso di transistori bipolari a giunzione (BJT) e resistenze. Si chiama logica transistor-transistor perché il transistor svolge sia la funzione logica che la funzione di inversione-amplificazione. I circuiti integrati basati sulla famiglia TTL sono stati ampiamente usati in applicazioni quali computer, controlli industriali, apparecchiature di prova, strumentazione elettronica, elettronica di consumo, sintetizzatori e molto altro. Dopo la loro introduzione come circuito integrato a opera di *Sylvania* nel 1963, i circuiti integrati TTL sono stati prodotti da diverse aziende di semiconduttori. La serie 7400 (chiamato anche 74xx) della *Texas Instruments* è diventata particolarmente popolare. I produttori di porte basati sulla tecnologia TTL hanno offerto una vasta gamma di porte logiche, flip-flop, contatori, multiplexer e altri circuiti.

Complementary Metal-Oxide-Semiconductor (CMOS) è la tecnologia più recente per la costruzione di porte logiche inserite in circuiti integrati. La tecnologia CMOS è usata in microprocessori, microcontrollori, nella RAM statica e in molti altri circuiti logici digitali. E' la tecnologia più raffinata, che consente un abbattimento dei consumi e dei tempi di risposta, congiuntamente alla possibilità di realizzare un'altissima densità di integrazione.

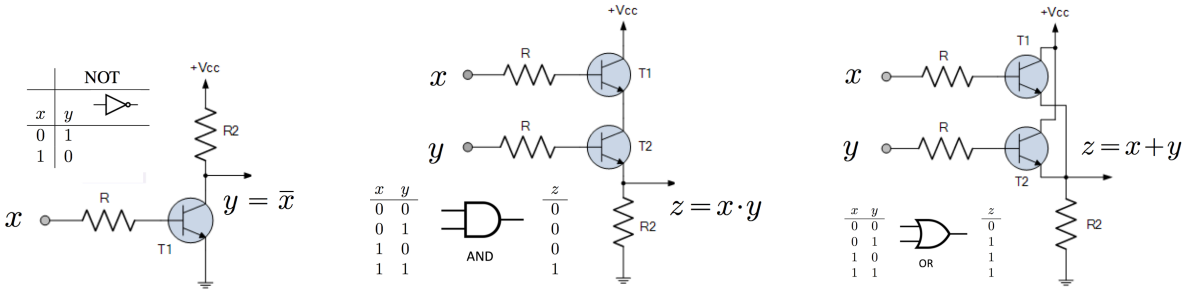
Non è questa la sede per descrivere, nel dettaglio, la realizzazione di tutte le porte logiche nelle varie tecnologie; ci limiteremo a una mera illustrazione didattica, che ha lo scopo essenziale di colmare la connessione che ancora manca tra porta logica astratta, che esprime una certa funzione Booleana, e la circuiteria che la realizza concretamente. Dati gli scopi limitati della nostra trattazione, faremo riferimento a circuiti assimilabili alla famiglia RTL, con qualche anticipazione sulle porte TTL e CMOS. Nei corsi di *Reti Logiche* ed *Elettronica* sarà poi possibile studiare meglio la struttura e il funzionamento delle porte più complesse.

NOT Nella porta NOT bisogna realizzare un'inversione del valore logico. Questa funzione si realizza mediante un singolo transistor, nel quale la variabile d'ingresso va ad alimentare la base, mentre quella di uscita si ricava sul collettore del transistor. Se associamo la costante 1 a un livello alto di tensione, p.es. la V_{CC} di figura 4.18a, e la costante 0 a un livello basso, cioè la tensione di massa pari a $0\,V$, il funzionamento dell'invertitore è il seguente: quando la variabile x assume valore logico 1, e cioè viene portata a tensione V_{CC}, il transistor si polarizza direttamente ed entra in piena conduzione. Tale condizione corrisponde alla saturazione vista in figura 3.25[2]; la tensione V_{CE} crolla idealmente a $0\,V$, portando la variabile y in uscita nello stato logico 0[3]. Quando viceversa la variabile x assume valore logico 0, e cioè viene portata a tensione $0\,V$ (connessione a massa), il transistor blocca la conduzione e la I_C diventa (praticamente) nulla. Tale condizione corrisponde all'interdizione vista in figura 3.25; la tensione V_{CE} diventa pari alla V_{CC}, portando a 1 logico il valore dell'uscita.

AND Nella porta AND bisogna fare in modo che se anche uno solo degli ingressi è a 0, l'uscita resti a 0. Il circuito di figura 4.18b realizza questa condizione. Poiché l'uscita a 1 significa livello alto di tensione, per realizzarla bisogna che entrambi i due transistor associati ai due ingressi, che sono posti in serie, siano in piena conduzione. Per realizzare ciò è necessario portare le variabili x e y d'ingresso a V_{CC}. Se anche uno solo dei due ingressi rimane a livello basso, uno dei due transistor va in interdizione e la tensione di uscita resta a livello basso, il che corrisponde a 0 logico.

[2] Per ottenere la saturazione bisogna dimensionare adeguatamente le resistenze R e R_2

[3] In realtà sussiste la tensione V_{CEsat}, di circa $0,3\,V$ per il germanio e $0,6\,V$ per il silicio

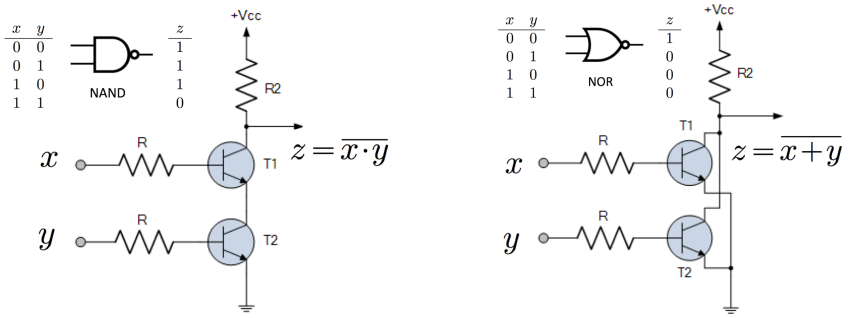


(a) Porta NOT (b) Porta AND (c) Porta OR

Figura 4.18: Porte NOT, AND e OR realizzate in tecnologia RTL

OR Nella porta OR bisogna fare in modo che se anche uno solo degli ingressi è a 1, l'uscita resti a 1. Il circuito di figura 4.18c realizza questa condizione. Poiché l'uscita a 1 significa livello alto di tensione, per realizzarla basta che anche uno solo dei due transistor associati ai due ingressi, che sono posti in parallelo, sia in piena conduzione. Per realizzare ciò è sufficiente portare o l'ingresso x o l'ingresso y al valore V_{CC}. Solo se entrambi gli ingressi rimangono a livello basso i transistor sono in interdizione e la tensione di uscita resta a livello basso, il che corrisponde a 0 logico.

NAND La porta NAND viene realizzata a partire dal circuito della porta AND, spostando semplicemente la resistenza $R2$ di carico dall'emettitore di $T2$ al collettore di $T1$ (si veda la figura 4.19a). In questo modo, quando entrambi i transistor sono in piena conduzione grazie al fatto che entrambi gli ingressi sono a 1, l'uscita va a 0; e questa rimane l'unica condizione per la quale si ha uscita bassa. Infatti se anche uno solo degli ingressi va a 0, uno dei due transistor va in interdizione e l'uscita resta a 1.



(a) Porta NAND realizzata con tecnologia RTL

(b) Porta NOR realizzata con tecnologia RTL

Figura 4.19: Porte NAND e NOR in tecnologia RTL

NOR Anche in questo caso si parte della porta negata, che è la OR, spostando la resistenza $R2$ di carico dall'emettitore di $T2 - T1$ al collettore di $T2 - T1$ (si veda la figura 4.19b). In questo modo, quando entrambi i transistor sono in interdizione grazie al fatto che entrambi gli ingressi sono a 0, l'uscita va a 1; e questa rimane l'unica condizione per la quale si ha uscita alta. Infatti se anche uno solo degli ingressi va a 1, uno dei due transistor va in saturazione e l'uscita collassa a 0.

4.3.4 Forme canoniche

Sia assegnata una qualunque funzione Booleana espressa mediante tavola di verità; tanto per fissare le idee possiamo prendere la funzione descritta dalla figura 4.6.

x	y	z	$f = (x+y)\cdot(\overline{x}+z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) La funzione Booleana $f = (x+y)\cdot(\overline{x}+z)$

x	y	z	$\overline{x}\,y\,\overline{z}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(b) Termine minimo $\overline{x}\,y\,\overline{z}$ associato a 010

Figura 4.20: Esempio di funzione Booleana e uno dei termini minimi a essa associati

Supponiamo ora di non sapere quale sia la sua espressione Booleana e di volerla, in qualche modo, ricavare. Osserviamo che la funzione vale 1 in corrispondenza delle terne 010, 011, 101 e 111, cioè quando in ingresso si presenta la terna 010, *oppure* la terna 011, *oppure* ... ecc. Questo ci consente di costruire la nostra funzione a partire dalla somma logica di funzioni *elementari*[4], cioè quelle funzioni che valgono 1 esattamente in corrispondenza di una e una sola configurazione d'ingresso, precisamente quella che entra nella somma logica; una tale funzione si chiama *termine minimo* (o *minterm*). Come esempio prendiamo la funzione che vale 1 solo in corrispondenza della terna 010; essa può essere espressa come prodotto logico di variabili dirette e negate, le prime in corrispondenza delle variabili che in ingresso valgono 1, le seconde in corrispondenza delle variabili che in ingresso valgono 0. Nel nostro caso si tratta della funzione $\overline{x}\,y\,\overline{z}$, rappresentata in figura 4.20b[5]. Un ragionamento analogo si può fare per gli tutti altri termini minimi che compaiono nella funzione f di figura 4.20a; così facendo si ottengono le funzioni $\overline{x}\,y\,z$ per 011, $x\,\overline{y}\,z$ per 101 e $x\,y\,z$ per 111. La funzione di figura 4.20a si può allora esprimere come *somma di prodotti* delle variabili d'ingresso, nella forma

$$f = \overline{x}\,y\,\overline{z} + \overline{x}\,y\,z + x\,\overline{y}\,z + x\,y\,z \tag{4.15}$$

La tecnica che abbiamo appena usato può essere generalizzata a qualunque funzione $y = f(x_1, x_2, \cdots, x_n)$ di n variabili, tenuto conto che i termini minimi m_i sono in tutto tanti quante sono le possibili configurazioni d'ingresso, cioè 2^n. Se codifichiamo le n-ple d'ingresso associate a ciascun termine minimo con il corrispondente intero rappresentato in notazione posizionale in base 2, possiamo indicare i termini minimi che compongono la sommatoria usando gli interi compresi tra 0 e $2^n - 1$. Tenendo conto di tutto ciò la generica funzione si rappresenta come

$$f(x_1, x_2, \cdots, x_n) = \sum_{i=0}^{2^n-1} \mu_i m_i = \sum_{i:\mu_i=1} m_i \tag{4.16}$$

dove μ_i è il valore assunto dalla funzione in corrispondenza del termine minimo m_i e $0 \leqslant i \leqslant 2^n-1$; nel nostro esempio si ha $m_2 = \overline{x}y\overline{z}$, $m_3 = \overline{x}yz$, $m_5 = x\overline{y}z$, $m_7 = xyz$, $\mu_2 = \mu_3 = \mu_5 = \mu_7 = 1$, $\mu_0 = \mu_1 = \mu_4 = \mu_6 = 0$. La formula (4.16) rappresenta la funzione nei termini della cosiddetta *prima forma canonica* (o *somma di prodotti*). Se nella tavola di verità della nostra funzione mettiamo in evidenza la codifica dei termini minimi otteniamo

[4] A rigore si tratterebbe di un OR esclusivo, visto che le configurazioni d'ingresso si escludono l'un l'altra, nel senso che se compare una non può comparire contemporaneamente anche l'altra. Tuttavia, visto che non è possibile avere in ingresso due configurazioni in contemporanea, i valori di XOR e OR coincidono, e quindi si fa riferimento a quest'ultima.

[5] tralasciamo d'ora in poi l'uso del $\cdot$ per semplificare la notazione

		x	y	z	f	
m_0	$\overline{x}\overline{y}\overline{z}$	0	0	0	0	μ_0
m_1	$\overline{x}\overline{y}z$	0	0	1	0	μ_1
m_2	$\overline{x}y\overline{z}$	0	1	0	1	μ_2
m_3	$\overline{x}yz$	0	1	1	1	μ_3
m_4	$x\overline{y}\overline{z}$	1	0	0	0	μ_4
m_5	$x\overline{y}z$	1	0	1	1	μ_5
m_6	$xy\overline{z}$	1	1	0	0	μ_6
m_7	xyz	1	1	1	1	μ_7

Figura 4.21: Codifica dei termini minimi

$$f(x,y,z) = \sum_{i\in\{2,3,5,7\}} m_i = m_2 + m_3 + m_5 + m_7 = \overline{x}y\overline{z} + \overline{x}yz + x\overline{y}z + xyz \tag{4.17}$$

poiché $2, 3, 5$ e 7 sono le codifiche in base due di 010, 011, 101 e 111.

Un discorso del tutto analogo si può fare procedendo per dualità, e realizzando un *prodotto di somme*. Ciò richiede di analizzare i termini per i quali la funzione va a 0; dalla figura 4.20a osserviamo che ciò accade in corrispondenza delle terne 000, 001, 100 e 110, cioè $0, 1, 4$ e 6 con la codifica usata prima. L'idea è quella di esprimere il valore della funzione come prodotto di somme che sono sempre a 1, tranne che per una singola configurazione che le manda a 0; una funzione che vale sempre 1, tranne che per una singola configurazione per la quale vale 0 si chiama *termine massimo* (o *maxterm*), e viene indicata con M_i. In figura 4.22 viene rappresentato M_1; il modo più semplice per esprimere un termine massimo è quello di ricorrere alla somma logica di variabili dirette e negate, le prime in corrispondenza delle variabili che in ingresso valgono 0, le seconde in corrispondenza delle variabili che in ingresso valgono 1. Nel caso di figura 4.22 si tratta della funzione $x + y + \overline{z}$

x	y	z	$x + y + \overline{z}$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figura 4.22: Termine massimo $x + y + \overline{z}$ associato a 001

Tenendo conto di tutto ciò si perviene alla *seconda forma canonica* (o *prodotti di somme*), nella quale la generica funzione si rappresenta come

$$f(x_1, x_2, \cdots, x_n) = \prod_{i=0}^{2^n-1} (\mu_i + M_i) = \prod_{i:\mu_i=0} M_i \tag{4.18}$$

dove $0 \leqslant i \leqslant 2^n - 1$, μ_i è il valore assunto dalla funzione in corrispondenza del termine massimo M_i, il quale si codifica secondo la procedura prima descritta; nel nostro esempio si ha $M_0 = x + y + z$, $M_1 = x + y + \overline{z}$, $M_4 = \overline{x} + y + z$, $M_6 = \overline{x} + \overline{y} + z$ e così via.
Se mettiamo in evidenza nella tavola di verità della nostra funzione la codifica dei termini massimi otteniamo

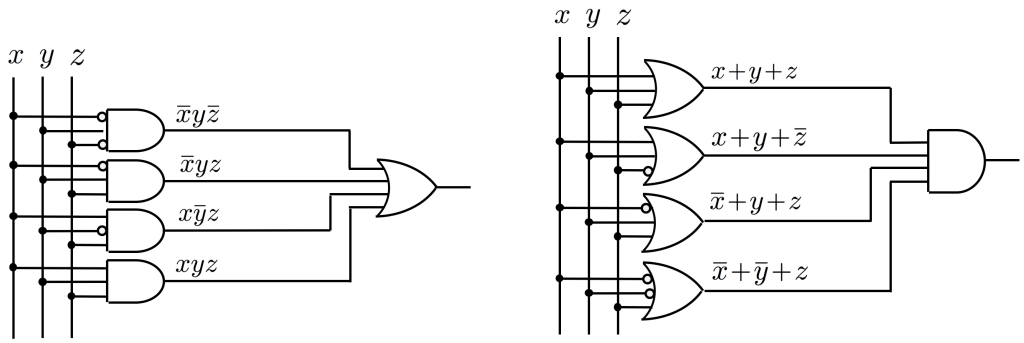
		x	y	z	f	
M_0	$x+y+z$	0	0	0	0	μ_0
M_1	$x+y+\overline{z}$	0	0	1	0	μ_1
M_2	$x+\overline{y}+z$	0	1	0	1	μ_2
M_3	$x+\overline{y}+\overline{z}$	0	1	1	1	μ_3
M_4	$\overline{x}+y+z$	1	0	0	0	μ_4
M_5	$\overline{x}+y+\overline{z}$	1	0	1	1	μ_5
M_6	$\overline{x}+\overline{y}+z$	1	1	0	0	μ_6
M_7	$\overline{x}+\overline{y}+\overline{z}$	1	1	1	1	μ_7

Figura 4.23: Codifica dei termini massimi

Esprimendo la funzione di figura 4.20a in questo modo si ottiene

$$f(x,y,z) = \prod_{i\in\{0,1,4,6\}} M_i = M_0 \cdot M_1 \cdot M_4 \cdot M_6 = (x+y+z)\cdot(x+y+\overline{z})\cdot(\overline{x}+y+z)\cdot(\overline{x}+\overline{y}+z) \quad (4.19)$$

poiché $0, 1, 4$ e 6 sono le codifiche in base due di 000, 001, 100 e 110.
La realizzazione operativa delle funzioni (4.17) e (4.19) mediante porte AND, OR, NOT è basata sui circuiti di figura 4.24. Poiché ogni porta AND, OR a n ingressi richiede $n-1$ porte a 2 ingressi, per il circuito di figura 4.24a



(a) Circuito derivante dai *minterm* relativi alla funzione di figura 4.21

(b) Circuito derivante dai *maxterm* relativi alla funzione di figura 4.23

Figura 4.24: Porte NOT, AND e OR necessarie a realizzare la funzione di figura 4.20a secondo *minterm* e *maxterm*

servono $2 \cdot 4 = 8$ porte AND e 3 porte OR a 2 ingressi; per l'altro circuito di figura 4.24b servono $2 \cdot 4 = 8$ porte OR e 3 porte AND a 2 ingressi; in entrambi i casi servono dunque 11 porte a 2 ingressi.
Naturalmente le espressioni (4.17) (somma di prodotti) e (4.19) (prodotto di somme) rappresentano la stessa funzione, come si può verificare facilmente. Prendiamo la somma dei prodotti e cerchiamo di semplificarla usando gli assiomi e i teoremi introdotti in precedenza nelle sezioni 4.2.1 e 4.2.2

$$\overline{x}y\overline{z} + \overline{x}yz + x\overline{y}z + xyz \overset{(A6)}{=} \overline{x}y(\overline{z}+z) + xz(\overline{y}+y) \overset{(A7)}{=} \overline{x}y\cdot 1 + xz\cdot 1 \overset{(A3)}{=} \overline{x}y + xz \quad (4.20)$$

Se invece prendiamo il prodotto di somme si ottiene

$$(x+y+z)\cdot(x+y+\overline{z})\cdot(\overline{x}+y+z)\cdot(\overline{x}+\overline{y}+z) \overset{(T9)}{=} (x+y)\cdot(\overline{x}+z) \overset{(T10)}{=} \overline{x}y + xz \quad (4.21)$$

A seguito della semplificazione la complessità del circuito associato alla funzione si riduce drasticamente, come si può vedere dalla figura 4.25; servono in tutto 3 porte a due ingressi al posto di 11. Le formule (4.17) e (4.19), che portano entrambe alla forma semplificata $\overline{x}y + xz$ per la funzione appena analizzata, aprono il problema

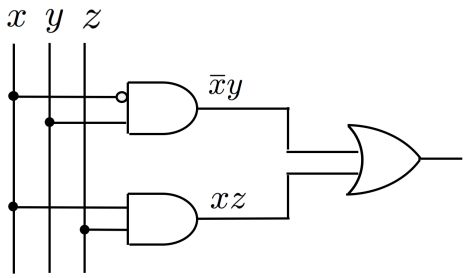


Figura 4.25: Circuito semplificato equivalente ai circuiti di figura 4.24

della ricerca della forma algebrica minima, che consenta cioè di realizzare la funzione usando il minimo numero possibile di porte logiche. Tanto per inquadrare la questione, si tenga conto che la (4.17) richiederebbe quattro porte AND con tre ingressi e una porta OR a quattro ingressi, più tutti i NOT che servono alla complementazione delle varie variabili; nel caso della (4.19) servono invece quattro porte OR con tre ingressi e una porta AND a quattro ingressi; se invece usiamo la funzione semplificata bastano due AND e un OR a due ingressi. Affronteremo nel seguito il problema della minimazione delle espressioni.
Si osservi inoltre che è sempre possibile passare dal prodotto di somme (4.21) alla somma di prodotti (4.20) e viceversa, sfruttando il T7 di De Morgan a partire dalla doppia negazione T3

$$\overline{\overline{(x+y+z)\cdot(x+y+\overline{z})\cdot(\overline{x}+y+z)\cdot(\overline{x}+\overline{y}+z)}} \overset{(T7)}{=} \overline{\overline{(x+y+z)}+\overline{(x+y+\overline{z})}+\overline{(\overline{x}+y+z)}+\overline{(\overline{x}+\overline{y}+z)}}$$
$$\overset{(T7)}{=} \overline{\overline{x}\,\overline{y}\,\overline{z}+\overline{x}\,\overline{y}\,z+x\,\overline{y}\,\overline{z}\;+\;x\,y\,\overline{z}}$$

La semplificazione si può a questo punto concludere nel modo seguente

$$\overline{\overline{x}\,\overline{y}\,\overline{z}+\overline{x}\,\overline{y}\,z+x\,\overline{y}\,\overline{z}\;+\;x\,y\,\overline{z}} \overset{(A6)}{=} \overline{\overline{x}\,\overline{y}(z+\overline{z})+x\,\overline{z}(y+\overline{y})} \overset{(A7)}{=} \overline{\overline{x}\,\overline{y}+x\,\overline{z}} \overset{(T7)}{=} \overline{\overline{x}\,\overline{y}}\cdot\overline{x\,\overline{z}}$$
$$\overset{(T7)}{=} (x+y)(\overline{x}+z) \overset{(T10)}{=} \overline{x}y+xz \tag{4.22}$$

In questo caso il passaggio ha richiesto molti calcoli in più, e non è stato conveniente; tuttavia la tecnica può soccorrere in qualche situazione particolare.
Quanto detto finora conferma nuovamente che ogni funzione Booleana $f(x_1, x_2, ..., x_n)$ può essere espressa indifferentemente o come somma di prodotti (OR di AND) o come prodotto di somme (AND di OR). Il teorema T7 di De Morgan costituisce lo strumento di connessione tra le due forme canoniche, consentendo di passare dall'una all'altra. La giustificazione formale è data dal fatto che possiamo esprimere la funzione f come la negazione di una certa funzione φ, espressa come somma dei termini minimi che corrispondono agli zeri della funzione f; in tal caso si ha $\varphi_i = \overline{\mu_i}$ e si ottiene

$$\overline{\varphi} = \overline{\sum_{i=0}^{2^n-1} \varphi_i m_i} = \overline{\sum_{i:\mu_i=0} \varphi_i m_i} = \prod_{i=0}^{2^n-1} (\overline{\varphi_i} + \overline{m_i}) = \prod_{i=0}^{2^n-1} (\mu_i + M_i) = f$$

nella quale la terza uguaglianza deriva dall'applicazione del teorema di De Morgan.
Per chiarire il procedimento possiamo prendere nuovamente come esempio la funzione di figura 4.21 studiata precedentemente

		x	y	z	f		φ	
m_0	$\overline{x}\overline{y}\overline{z}$	0	0	0	0	μ_0	1	φ_0
m_1	$\overline{x}\overline{y}z$	0	0	1	0	μ_1	1	φ_1
m_2	$\overline{x}y\overline{z}$	0	1	0	1	μ_2	0	φ_2
m_3	$\overline{x}yz$	0	1	1	1	μ_3	0	φ_3
m_4	$x\overline{y}\overline{z}$	1	0	0	0	μ_4	1	φ_4
m_5	$x\overline{y}z$	1	0	1	1	μ_5	0	φ_5
m_6	$xy\overline{z}$	1	1	0	0	μ_6	1	φ_6
m_7	xyz	1	1	1	1	μ_7	0	φ_7

Figura 4.26: La funzione di figura 4.21 con i valori della φ_i

Si ricava allora

$$f = \sum_{i:\mu_i=1} \mu_i m_i = \mu_2 m_2 + \mu_3 m_3 + \mu_5 m_5 + \mu_7 m_7 = \overline{x}y\overline{z} + \overline{x}yz + x\overline{y}z + xyz \tag{4.23}$$

$$\overline{\varphi} = \overline{\sum_{i:\mu_i=0} \varphi_i m_i} = \overline{\varphi_0 m_0 + \varphi_1 m_1 + \varphi_4 m_4 + \varphi_6 m_6} = \overline{\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + x\,\overline{y}\,\overline{z} \;+\; x\,y\,\overline{z}} \tag{4.24}$$

$$= \overline{\overline{x}\,\overline{y}\,\overline{z}} \cdot \overline{\overline{x}\,\overline{y}\,z} \cdot \overline{x\,\overline{y}\,\overline{z}} \cdot \overline{x\,y\,\overline{z}} = (x+y+z)\cdot(x+y+\overline{z})\cdot(\overline{x}+y+z)\cdot(\overline{x}+\overline{y}+z)$$

dove la 4.23 è espressa mediante *minterm*, mentre la 4.24 è espressa mediante *maxterm*. Siamo dunque passati dalla somma dei prodotti al prodotto delle somme.

4.3.5 Interpretazione circuitale

Come già accennato in precedenza, la prima applicazione circuitale dell'Algebra Booleana si deve a Shannon, che la usò nella progettazione di circuiti complessi per la commutazione a contatti. Questo tipo di applicazione è ancora largamente usata, anche se i contatti dei relè sono oggi sostituiti da dispositivi a stato solido (SCR, TRIAC). Tuttavia l'importanza dell'Algebra Booleana è legata soprattutto all'impiego nell'ambito dei circuiti logici, il cui peso è oggidì preponderante.
Nell'interpretazione di Shannon le costanti logiche 0 e 1 indicano rispettivamente un circuito aperto o uno chiuso, mentre le variabili indicano il contatto di un interruttore o di un relè. Con i simboli x e $\overline{x}$ si indicano due contatti azionati contemporaneamente, ma sempre tali che quando uno è aperto, l'altro è chiuso e viceversa. Si consideri ora un assieme di contatti $x_1, x_2, \cdots, x_n$ in parallelo tra loro, come illustrato in figura 4.27

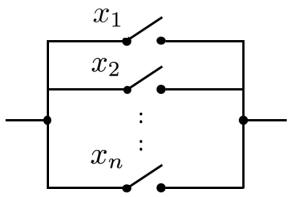


Figura 4.27: n contatti in parallelo realizzano la funzione Booleana $x_1 + x_2 + \cdots + x_n$

Il circuito presenterà continuità elettrica tra i punti a e b quando almeno uno dei contatti è chiuso. Ne consegue che la somma logica

$$x_1 + x_2 + ... + x_n$$

descrive, secondo l'analogia di Shannon, il comportamento elettrico di n contatti in parallelo.
Se invece i contatti $x_1, x_2, \cdots, x_n$ sono in serie tra loro, come illustrato in figura 4.28

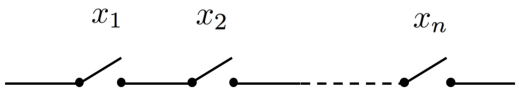


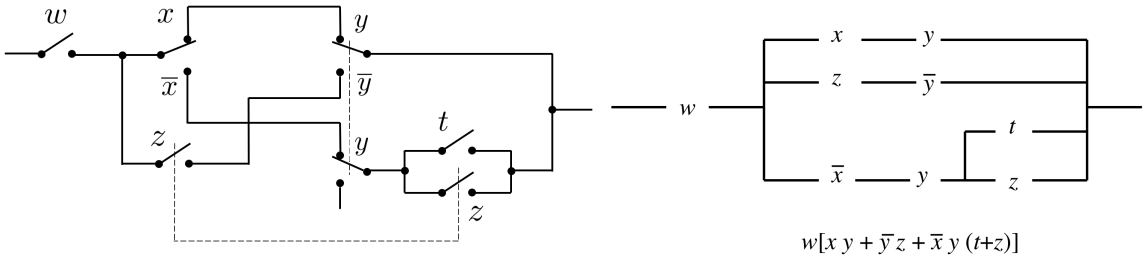
Figura 4.28: n contatti in serie realizzano la funzione Booleana $x_1 \cdot x_2 \cdot ... \cdot x_n$

il circuito presenterà continuità elettrica tra i punti a e b solo quando tutti i contatti sono chiusi. Ne consegue che il prodotto logico

$$x_1 \cdot x_2 \cdot ... \cdot x_n$$

descrive, secondo l'analogia di Shannon, il comportamento elettrico di n contatti in serie. L'associazione tra stato di apertura di un contatto e variabile Booleana apre alla possibilità di progettare reti complesse ricorrendo all'Algebra Booleana.

Esempio 4.1. Come primo esempio possiamo analizzare il circuito di figura 4.29a; esso è costituito da alcuni interruttori e due *deviatori*; un deviatore è un interruttore nel quale si aggiunge un contatto anche per la posizione di riposo, in modo che possa realizzare continuità elettrica lungo due percorsi alternativi. Per prima cosa dobbiamo assegnare le variabili Booleane, tenendo conto che le linee tratteggiate che collegano gli interruttori indicano che essi sono azionati contemporaneamente dallo stesso pulsante. Fatta questa operazione si perviene al circuito di



(a) Circuito complesso di commutazione

(b) Variabili Booleane associate al circuito di figura 4.29a

Figura 4.29: Circuito di commutazione e sua rappresentazione Booleana

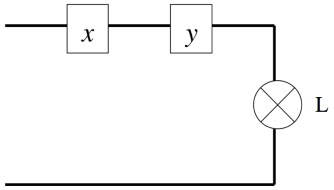
figura 4.29b. Esso è costituito dalla serie tra la variabile w e un blocco circuitale, costituito da tre rami in parallelo; ecco allora che avremo come espressione Booleana associata $w \cdot (p_1 + p_2 + p_3)$, dove p_1, p_2 e p_3 sono i tre rami in parallelo; p_1 è la serie di x e y, e dunque $p_1 = xy$, mentre $p_2 = z\overline{y}$. p_3 è invece la serie di tre elementi, l'ultimo dei quali è il parallelo tra t e z; dunque $p_3 = \overline{x}y(t + z)$. Mettendo tutto assieme si ottiene

$$w \cdot [xy + z\overline{y} + \overline{x}y(t + z)]$$

○

Questo appena fatto è un esempio di analisi di un circuito. E' evidente che a partire dall'espressione Booleana potremmo ricavare la tavola di verità e sapere qual è il comportamento del circuito. Ma l'Algebra Booleana eprime tutte le sue enormi potenzialità soprattutto quando si vuole progettare un circuito che esegua determinate funzioni, e cioè quando bisogna realizzare la *sintesi* di un circuito.

Esempio 4.2. Come esempio concreto, immaginiamo si voglia risolvere il seguente problema, di interesse molto pratico. In un appartamento c'è un corridoio; quando si entra si vuole accendere la luce dall'interruttore x prossimo all'ingresso, che supponiamo sia a destra nel disegno di figura 4.31); uscendo si vuole spegnere la luce dall'interruttore y prossimo all'uscita sulla sinistra. Per risolvere il problema, dobbiamo esprimere le condizioni



(a) Gli interruttori x e y comandano la luce di un corridoio

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

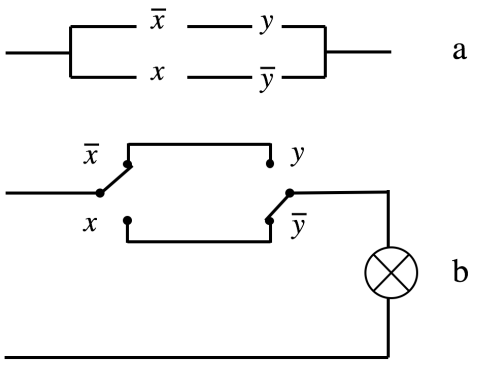
(b) Tavola di verità che esprime il funzionamento del circuito di figura 4.31

Figura 4.30: Circuito che comando una luce da due punti e relativa tavola di verità

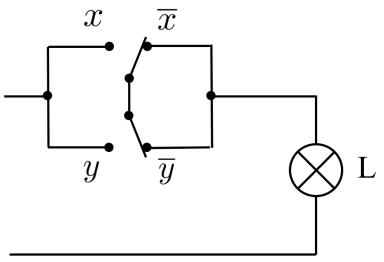
di funzionamento della lampada e degli interruttori ricorrendo agli strumenti dell'Algebra Booleana. Ciascuno dei due interruttori può stare in una sola tra due posizioni, che associamo alle due variabili 0 e 1. Anche la luce L può essere spenta (0) o accesa (1). Ecco allora che dobbiamo costruire una tavola di verità con due variabili d'ingresso (x e y) e una variabile L che assumerà dei valori che sono *funzione* di x e y; abbiamo insomma una funzione Booleana $f : \mathbf{2^2} \to \mathbf{2}$.
Supponiamo che, entrando in corridoio con la luce ancora spenta, entrambi gli interruttori siano (p.es) nello stato 0. A questa configurazione d'ingresso corrisponde $L = 0$. Se entriamo in corridoio da sinistra, all'atto di accendere la luce cambiamo la posizione dell'interruttore x, che passa da 0 a 1; in corrispondenza la luce deve accendersi, e dunque alla coppia 01 corrisponde $L = 1$. Uscendo dal corridoio a destra si muove l'interruttore y, che comanda lo spegnimento della luce; e dunque a 11 corrisponde $L = 0$. La configurazione che manca, 01, è quella che si crea quando si entra in corridoio da destra con la luce spenta e si muove l'interruttore; dunque a 01 corrisponde $L = 1$. Se mettiamo tutto nella tavola di verità otteniamo la funzione di figura 4.30b. Usando la prima forma canonica ricaviamo immediatamente l'espressione risolutiva

$$x\overline{y} + \overline{x}y \tag{4.25}$$

la quale, nella nuova interpretazione circuitale, corrisponde al parallelo di due interruttori posti in serie, così come rappresentato in figura 4.31a.a; la figura 4.31a.b mostra invece l'attuazione pratica, che usa due deviatori; essi realizzano la condizione che se x è chiuso, necessariamente $\overline{x}$ deve essere aperto e viceversa. Naturalmente il



(a) Gli interruttori x e y che risolvono il problema dell'accensione della luce nel corridoio



(b) Altra configurazione circuitale, basata sulla II forma canonica

Figura 4.31: Circuiti che risolvono il problema del comando di una luce da due punti diversi

problema può essere risolto anche mediante la seconda forma canonica, usando la quale si ottiene

$$(x + y) \cdot (\overline{x} + \overline{y})$$

Questa impostazione porta a un circuito un po' strano (si veda la figura 4.31b), che funziona altrettanto correttamente, ma che non si usa nella pratica dei collegamenti elettrici poiché richiede una cavo in più per la connessione tra i due deviatori.
L'equazione 4.25, risolutiva del problema, corrisponde anche al circuito di uno XOR, già visto nella figura 4.11b. In che rapporto sta allora quest'ultimo con il circuito 4.31a? La realizzazione mediante interruttori consente di incarnare la funzione Booleana direttamente al livello base dei circuiti di commutazione, in modo tale che ogni variabile Booleana sia immersa nel circuito; quella basata sulla porta XOR costituisce invece una sorta di interfaccia tra variabili d'ingresso, che determinano il comportamento del circuito, e la variabile d'uscita, che deve comandare la lampadina. Questo secondo approccio è concettualmente più elegante, ma soprattutto consente di svincolare il carico dalla rete logica. Se infatti il carico assorbe molta corrente, tutti gli interruttori della rete dovranno essere dimensionati di conseguenza; ciò potrebbe essere poco funzionale, visto che gli interruttori in grado di gestire correnti maggiori sono più grossi, costosi e più difficili da azionare. Usando invece un circuito simile a quello di figura 4.32 solo l'interruttore del relè deve essere dimensionato in modo tale da poter gestire la corrente del carico; gli altri due deviatori, relativi ai due comandi x e y, possono essere anche a bassa corrente. Si noti che nel circuito l'uscita della porta OR controlla direttamente la base di un transistor; quando l'uscita OR va a 1, il transistor entra in saturazione, la V_{CE} crolla al valore di saturazione e tutta la tensione V_{CC} di alimentazione va sul relè, che commutando chiude il circuito d'uscita e accende la lampadina. ◯

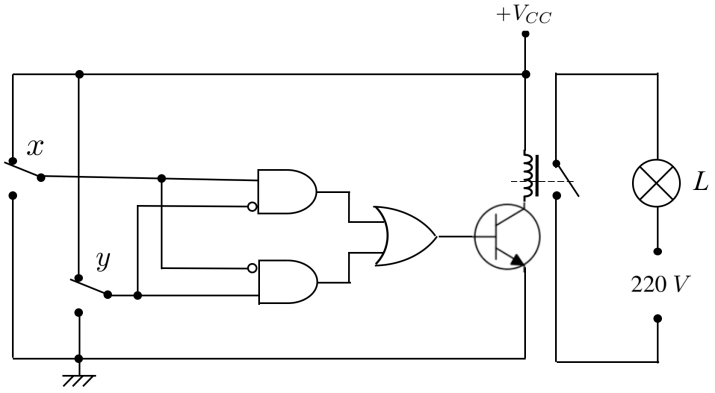


Figura 4.32: Controllo di una lampadina di potenza mediante una rete logica asservita a un transistor che controlla un relè

Esempio 4.3. Proviamo ora a complicare un po' il progetto, portando a tre il numero degli interruttori. Si vuol progettare un circuito capace di accendere o di spegnere una lampada mediante uno qualsiasi di tre interruttori indipendenti. Indicando al solito con $L = 1$ la condizione di lampada accesa, fissiamo come specifica che quando i tre interruttori sono aperti L valga 0. Chiudendo uno qualsiasi degli interruttori L dovrà assumere il valore 1, mentre tornerà al valore 0 azionando un qualsiasi altro interruttore. Infine essa assumerà nuovamente il valore 1 quando tutti i tre interruttori saranno chiusi. Anche questa situazione è molto frequente in pratica, per esempio in un ampio salone nel quale si vogliano avere più punti luce. Chiamando x, y, z le variabili logiche associate a ciascun interruttore si ricava la tavola di verità di figura 4.33

x	y	z	L
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figura 4.33: Tavola di verità per il circuito a tre interruttori

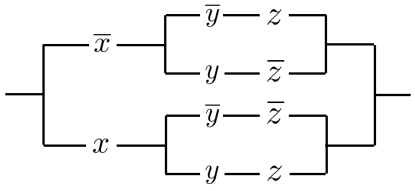
La sintesi del circuito mediante termini minimi fornisce

$$\overline{x}\,\overline{y}\,z + \overline{x}\,y\,\overline{z} + x\,\overline{y}\,\overline{z} + x\,y\,z$$

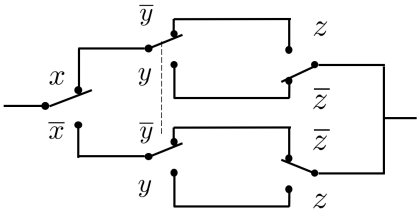
Prima di procedere alla realizzazione circuitale, è però opportuno semplificare il più possibile l'espressione, ottenendo

$$\overline{x}\,(\overline{y}\,z + y\,\overline{z}) + x(\overline{y}\,\overline{z} + y\,z)$$

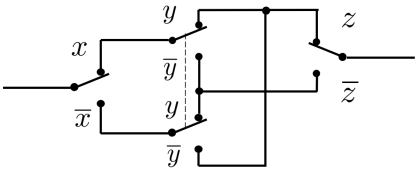
Il circuito che si ricava è rappresentato in figura 4.34.



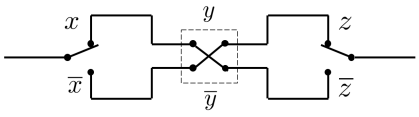
(a) Schema astratto per un controllo luce mediante tre interruttori basato sulla somma di prodotti



(b) Configurazione circuitale corrispondente



(c) Semplificazione circuitale del circuito 4.34b



(d) Attuazione pratica, mediante invertitore, del circuito 4.34c

Figura 4.34: Circuito con tre interruttori ottenuto da una sintesi basata sui termini minimi: realizzazione teorica e circuito commerciale, facente uso di un invertitore

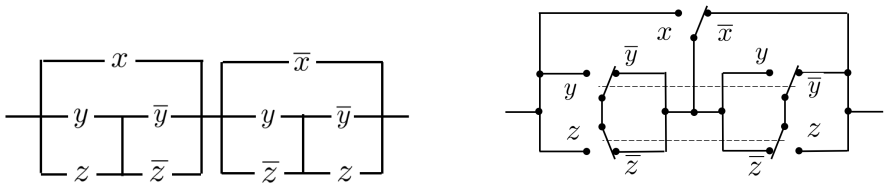
Nella figura 4.34a viene rappresentato lo schema astratto, mentre nella 4.34b si illustra il circuito corrispondente. Si osservi tuttavia che i due interruttori associati alla variabile $\overline{z}$, quando sono chiusi (come in figura) portano allo stesso potenziale le variabili y e $\overline{y}$ della parte centrale del circuito; dunque i due rami più interni si possono fondere. Poiché lo stesso discorso vale anche per gli interruttori della variabile z, il risultato finale è che si semplifica il circuito e si risparmia un deviatore. Il circuito semplificato è rappresentato in figura 4.34c, mentre in figura 4.34d si mostra la realizzazione commerciale, che fa uso del cosiddetto *invertitore*[6], rappresentato nello schema con le connessioni ingresso-uscita incrociate; l'altra posizione è quella con le connessioni dirette. L'invertitore consente

[6]Tale invertitore nulla ha a che fare con l'altro invertitore introdotto nell'algebra Booleana

fra l'altro di generalizzare il problema a $n \geqslant 4$ interruttori; in tutti questi casi il circuito risolutivo è dato da due deviatori in testa alla catena e da $n-2$ invertitori in cascata nelle posizioni intermedie.
Se ora passiamo a una sintesi mediante termini massimi otteniamo

$$(x+y+z)(x+\overline{y}+\overline{z})(\overline{x}+y+\overline{z})(\overline{x}+\overline{y}+z) = [x+(y+z)(\overline{y}+\overline{z})]\cdot[(\overline{x}+(y+\overline{z})(\overline{y}+z)]$$

che con la semplificazione porta al circuito di figura 4.35. Anche in questo caso la soluzione basata sui termini massimi è meno vantaggiosa, poiché richiede cinque deviatori invece di quattro. ◯



(a) Schema astratto per un controllo luce mediante tre interruttori basato sul prodotto di somme

(b) Configurazione circuitale corrispondente

Figura 4.35: Circuito con tre interruttori ottenuto da una sintesi basata sui termini massimi

Al di là della straordinaria importanza applicativa del metodo formale basato sull'Algebra Booleana, è importante riflettere, soprattutto alla luce di quest'ultimo esempio, che molto spesso in ambito ingegneristico la teoria offre diverse soluzioni tra loro equivalenti; sta poi al progettista scegliere la migliore, tenuto conto della razionalità e semplicità delle soluzioni prospettate, dei costi, del numero di componenti impiegati ecc.

4.3.6 Semplificazione delle espressioni Booleane

Si è già visto, dagli esempi che precedono, che le forme canoniche non esauriscono le espressioni analitiche di una funzione; anzi, come per qualsiasi relazione algebrica, anche quelle logiche possono essere trasformate in un certo numero di espressioni formalmente diverse, ma equivalenti dal punto di vista matematico. Ad esempio

$$\begin{aligned} f &= \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + \overline{x}\,y\,z + x\,y\,z \\ &= \overline{x}\,\overline{y}(\overline{z}+z) + y\,z(\overline{x}+x) = \overline{x}\,\overline{y} + y\,z \end{aligned}$$

Si diranno *equivalenti* due funzioni che abbiano la stessa tavola di verità, forma semplificata di una funzione ogni sua espressione non canonica, forma minima quella in cui ogni variabile, diretta o negata che sia, compare il minor numero di volte.
Espressioni semplificate si possono ottenere applicando le relazioni fondamentali dell'algebra Booleana alle espressioni canoniche, ma questa strada richiede una notevole pratica, si applica facilmente solo a funzioni di un limitato numero di variabili e non dà alcuna garanzia di pervenire effettivamente alla forma minima, dato il carattere artigianale del procedimento. Si semplifichi per esempio la funzione:

$$f = x\,y\,z\,w + x\,y\,z\,\overline{w} + x\,y\,\overline{z}\,w + \overline{x}\,y\,z\,w + x\,y\,\overline{z}\,\overline{w}$$

Per la proprietà distributiva il primo termine si può semplificare con il secondo, mentre il terzo termine si può semplificare con il quinto, ottenendo:

$$\begin{aligned} f &= x\,y\,z\,(w+\overline{w}) + x\,y\,\overline{z}\,(w+\overline{w}) + \overline{x}\,y\,z\,w = x\,y\,z\ + x\,y\,\overline{z}\ + \overline{x}\,y\,z\,w \\ &= x\,y + \overline{x}\,y\,z\,w = y\,(x+\overline{x}\,z\,w) = y\,(x+z\,w) \end{aligned}$$

nella quale l'ultima uguaglianza deriva dal secondo teorema dell'assorbimento T5.
Il termine

$$x\,y\,z = x\,y\,z\,w + x\,y\,z\,\overline{w}$$

viene detto *implicante*, in quanto implica i termini $x\,y\,z\,w$ e $x\,y\,z\,\overline{w}$, nel senso che vale 1 quando valgono 1 o l'uno o l'altro dei termini minimi implicati.

La semplificazione prodotta, nonostante sia efficace, deriva da una procedura arbitraria e non sistematica. Nel seguito studieremo due metodi procedurali per ottenere una semplificazione efficace.

Le mappe di Karnaugh

Il metodo proposto da *Karnaugh* è un metodo grafico di semplificazione che permette di ottenere molto semplicemente la forma minima di una funzione espressa come somma di prodotti *minterm*, facendo ricorso a particolari mappe di rappresentazione. Quale limitazione si ha che, sebbene il metodo sia concettualmente applicabile a funzioni di qualsiasi numero di variabili, esso diviene difficoltoso già per $5-6$ variabili.
Le mappe di Karnaugh, che possono essere considerate un ulteriore metodo di rappresentazione di una funzione logica, consistono in matrici m righe e k colonne, in cui $m = 2^i, k = 2^j$ per qualche i, j, col vincolo che $2^i \cdot 2^j = 2^n$ è pari al numero di elementi della matrice, e n è il numero delle variabili. Di conseguenza esse hanno 4 elementi per le funzioni di due variabili, 8 per quelle di tre variabili, 16 per quelle di quattro e così via. Ogni elemento della matrice rappresenta un termine minimo, che entra in un'espressione *minterm* di somma di prodotti. Nella figura 4.36 sono riprodotte le strutture delle mappe per $2, 3$ e 4 variabili.

$y \backslash x$	0	1
0	$\overline{x}\overline{y}$	$x\overline{y}$
1	$\overline{x}y$	xy

$z \backslash xy$	00	01	11	10
0	$\overline{x}\overline{y}\overline{z}$	$\overline{x}y\overline{z}$	$xy\overline{z}$	$x\overline{y}\overline{z}$
1	$\overline{x}\overline{y}z$	$\overline{x}yz$	xyz	$x\overline{y}z$

$zw \backslash xy$	00	01	11	10
00	$\overline{x}\overline{y}\overline{z}\overline{w}$	$\overline{x}y\overline{z}\overline{w}$	$xy\overline{z}\overline{w}$	$x\overline{y}\overline{z}\overline{w}$
01	$\overline{x}\overline{y}\overline{z}w$	$\overline{x}y\overline{z}w$	$xy\overline{z}w$	$x\overline{y}\overline{z}w$
11	$\overline{x}\overline{y}zw$	$\overline{x}yzw$	$xyzw$	$x\overline{y}zw$
10	$\overline{x}\overline{y}z\overline{w}$	$\overline{x}yz\overline{w}$	$xyz\overline{w}$	$x\overline{y}z\overline{w}$

Figura 4.36: Mappe di Karnaugh per 2, 3 e 4 variabili

Prendiamo ora come esempio il caso della funzione a 3 variabili descritta dalla tavola di verità di figura 4.37.

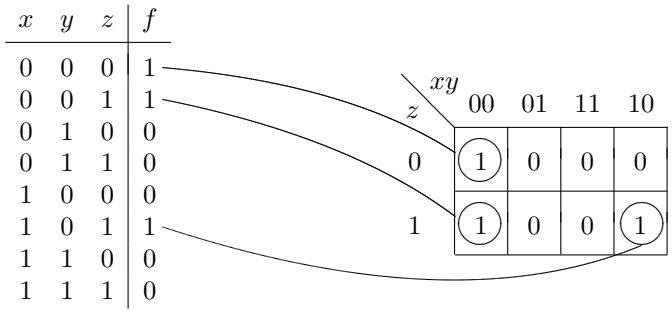


Figura 4.37: Costruzione di una mappa di Karnaugh per una funzione a 3 variabili

Usando la prima forma canonica *minterm* si ottiene

$$\overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z + x\overline{y}z$$

Nella mappa mettiamo un 1 in corrispondenza delle terne per le quali la funzione vale 1 e uno 0 in corrispondenza delle terne per le quali la funzione vale 0. L'assegnazione delle coordinate delle caselle della tabella dev'essere tale che passando da ciascuna casella a una adiacente, sia in senso orizzontale che verticale, vari il valore di una sola variabile. Si noti che devono essere considerate adiacenti anche le caselle terminali, come se la mappa fosse chiusa circolarmente su sè stessa, sia in senso orizzontale che verticale.
Ricapitolando si può dire che ciascun elemento della matrice corrisponde a un termine minimo di n variabili, e che la rappresentazione di una qualsiasi funzione di n variabili si ottiene contrassegnando con 1 o con 0 le posizioni corrispondenti ai termini minimi da cui la funzione è composta.
Una funzione riportata sulla mappa di Karnaugh può essere semplificata osservando che due caselle adiacenti, sia in senso orizzontale che verticale, differiscono per il valore di una sola variabile, che in una delle due caselle apparirà come variabile affermata, nell'altra come negata; si ricava dunque

$$fx + f\overline{x} = f(x + \overline{x}) = f$$

Per spiegare la procedura facciamo riferimento alla figura 4.38

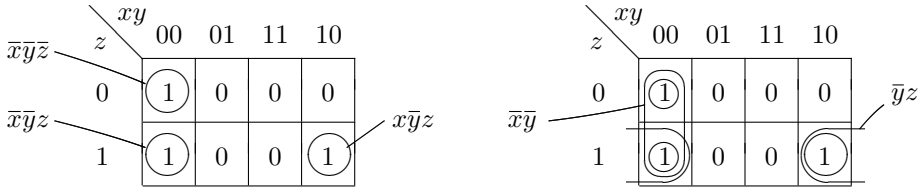


Figura 4.38: Semplificazione di una funzione mediante mappa di Karnaugh per una funzione a 3 variabili

Le caselle relative ai termini $\overline{x}\overline{y}\overline{z}$ e $\overline{x}\overline{y}z$ sono adiacenti, e la semplificazione è

$$\overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z = \overline{x}\overline{y}(z + \overline{z}) = \overline{x}\overline{y}$$

Sono però adiacenti anche le caselle $\overline{x}\overline{y}z$ e $x\overline{y}z$, che si semplificano come

$$\overline{x}\overline{y}z + x\overline{y}z = (\overline{x} + x)\overline{y}z = \overline{y}z$$

La funzione semplificata si ottiene allora come somma degli implicanti necessari e sufficienti a coprire tutti i *minterm* della funzione; nel caso di cui sopra si ha

$$f = \overline{x}\overline{y} + \overline{y}z$$

Facciamo ora un altro esempio, riconsiderando la funzione a tre variabili di figura 4.21, descritta dall'equazione (4.17) che riportiamo qua sotto per comodità del lettore

$$f(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}z + xyz \tag{4.26}$$

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

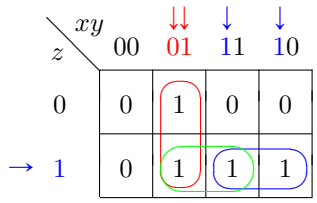


Figura 4.39: Mappa di Karnaugh per la funzione a 3 variabili descritta dalla funzione di figura 4.21

Se chiamiamo *sottocubo* il ricoprimento associato a un implicante, la regola pratica per individuare l'implicante a esso associato è che esso è costituito dal prodotto dalle variabili che rimangono costanti su 0 o su 1 all'interno del sottocubo stesso; nel primo dei due casi di figura 4.39 esse sono $x = 0$ e $y = 1$ (frecce rosse), e dunque $\bar{x}y$; nel secondo caso sono $x = 1$ e $z = 1$ (frecce blu), e dunque xz. Poiché i sottocubi devono avere 2^i elementi, per qualche valore di i, si possono costruire solo i due sottocubi con $i = 1$ che prendono dentro gli implicanti $\bar{x}y$ e xz; si osservi che il sottocubo associato all'implicante yz (in verde in figura), pur legittimo, sarebbe superfluo, poiché entrambi i suoi termini minimi $\bar{x}yz$ e xyz sono già compresi negli altri due sottocubi; d'altra parte già sappiamo, dal III teorema dell'assorbimento T6, che

$$\bar{x}y + xz + yz = \bar{x}y + xz \tag{4.27}$$

Facciamo ora un altro esempio, modificando la funzione nel modo seguente

$$f(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz \tag{4.28}$$

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

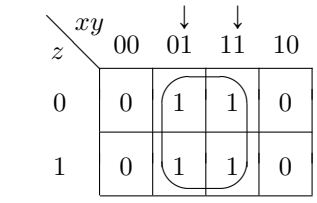


Figura 4.40: Altro esempio di mappa di Karnaugh per una funzione a 3 variabili

In questo caso si riesce a formare un sottocubo di 2^2 termini minimi, che hanno come implicante

$$\bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz = y(\bar{x}\bar{z} + \bar{x}z + x\bar{z} + xz) = y(\bar{x}(\bar{z} + z) + x(\bar{z} + z)) = y(\bar{x} + x) = y$$

Si osservi dalla figura 4.43 che $y = 1$ è in effetti l'unica variabile a rimanere costante.
Se ne deduce che un implicante di $n - 1$ variabili implica i termini minimi associati a 2 caselle adiacenti, un implicante di $n - 2$ variabili implica i termini minimi associati a 2^2 caselle adiacenti (appartenenti alla stessa riga o alla stessa colonna o raccolti attorno allo stesso vertice) e in generale un implicante di $n - i$ variabili implica i termini minimi associati a un sottocubo di 2^i caselle adiacenti. Un implicante si dice *primo* se corrisponde a un sottocubo non completamente coperto da altri sottocubi, *massimale* se non è possibile aumentare ulteriormente la sua dimensione; si definisce infine *copertura* della funzione un insieme di sottocubi tali da coprire tutti gli 1 della funzione. La minimizzazione della funzione si ricava a partire da una copertura minima, cioè una copertura formata dal più piccolo insieme di sottocubi primi massimali.

Facciamo ora un esempio con 4 variabili; la funzione

$$y = \overline{x}\,\overline{y}\,\overline{z}\,\overline{w} + \overline{x}\,y\,z\,w + \overline{x}\,\overline{y}\,z\,w + \overline{x}\,y\,z\,\overline{w} \tag{4.29}$$

è rappresentata dalla mappa di Karnaugh di figura 4.41.

zw \ xy	00	01	11	10
00	(1)	0	0	0
01	0	0	0	0
11	1	1	0	0
10	0	1	0	0

Figura 4.41: Mappa di Karnaugh per la funzione (4.29) di 4 variabili

I tre implicanti primi sono $\overline{x}\overline{y}\overline{z}\overline{w}$, $\overline{x}zw$ e $\overline{x}yz$; la loro somma rappresenta la funzione in forma semplificata.

Consideriamo ora la seguente funzione:

$$f = \overline{x}\,\overline{y}\,\overline{z}\,\overline{w} + \overline{x}\,\overline{y}\,\overline{z}\,w + \overline{x}\,\overline{y}\,z\,w + \overline{x}\,\overline{y}\,z\,\overline{w} + \overline{x}\,y\,\overline{z}\,w + \overline{x}\,y\,z\,w + x\,\overline{y}\,\overline{z}\,\overline{w} + x\,\overline{y}\,\overline{z}\,w + x\,\overline{y}\,z\,w + x\,\overline{y}\,z\,\overline{w} \tag{4.30}$$

In figura 4.42 è riportata la mappa di Karnaugh relativa ed è anche indicata la copertura minima di implicanti primi massimali.

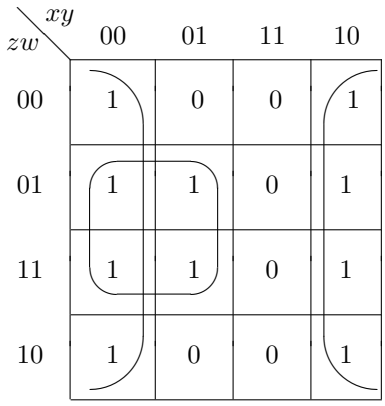


Figura 4.42: Mappa di Karnaugh per la funzione (4.30) di 4 variabili

La funzione semplificata risulta

$$y = \overline{x}\, w + \overline{y} \tag{4.31}$$

Se anziché considerare ciascuna casella della mappa come rappresentativa di un termine minimo la si considera rappresentativa di un termine massimo, la mappa può essere usata anche per costruire la funzione semplificata secondo la II forma canonica basata sul prodotto di somme. In tal caso i sottocubi e la relativa copertura minima vanno costruiti sugli 0. Ciascun sottocubo sarà l'implicante dei relativi termini massimi e verrà rappresentato in forma simbolica dalla somma logica delle variabili che rimangono costanti sul sottoinsieme, dirette se le relative coordinate valgono 0, negate se valgono 1. Tutto ciò deriva dalla circostanza che

$$(f + x)(f + \overline{x}) = f + x \cdot \overline{x} = f$$

Come esempio riprendiamo la funzione 4.26 e la relativa mappa

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

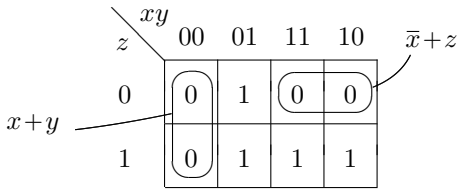


Figura 4.43: Mappa di Karnaugh usata per la semplificazione dei *maxterm*

La funzione semplificata che ne deriva è

$$(x + y)(\overline{x} + z) = xz + \overline{x}y$$

che corrisponde alla funzione 4.27 già trovata in precedenza basandoci sulla mappa di Karnaugh associata ai *minterm*.

Il metodo delle mappe di Karnaugh può essere applicato, come già accennato, anche a funzioni di 5 o 6 variabili. In tal caso non si potrà più realizzare una mappa piana, in cui ogni casella sia adiacente a caselle le cui coordinate differiscano per un'unica variabile. La mappa per cinque variabili viene pertanto realizzata mediante due mappe da quattro variabili, una associata al valore 0 della quinta variabile, l'altra al valore 1. Le adiacenze vanno quindi ricercate anche tra caselle occupanti posizioni omologhe sulle due mappe.
Si voglia ad esempio semplificare la seguente funzione

$$\begin{aligned} f = {} & x\,\overline{y}\,z\,w\,r + x\,\overline{y}\,z\,w\,\overline{r} + x\,\overline{y}\,z\,\overline{w}\,r + x\,\overline{y}\,z\,\overline{w}\,\overline{r} + \\ & + x\,\overline{y}\,\overline{z}\,w\,r + x\,\overline{y}\,\overline{z}\,w\,\overline{r} + x\,\overline{y}\,\overline{z}\,\overline{w}\,r + x\,\overline{y}\,\overline{z}\,\overline{w}\,\overline{r} + \\ & + \overline{x}\,y\,z\,w\,r + x\,y\,z\,w\,r + \overline{x}\,\overline{y}\,z\,w\,r + \overline{x}\,y\,\overline{z}\,\overline{w}\,r \end{aligned} \tag{4.32}$$

Le due mappe da 4 variabili, che si ottengono, una per $r = 0$ e una per $r = 1$ sono le seguenti

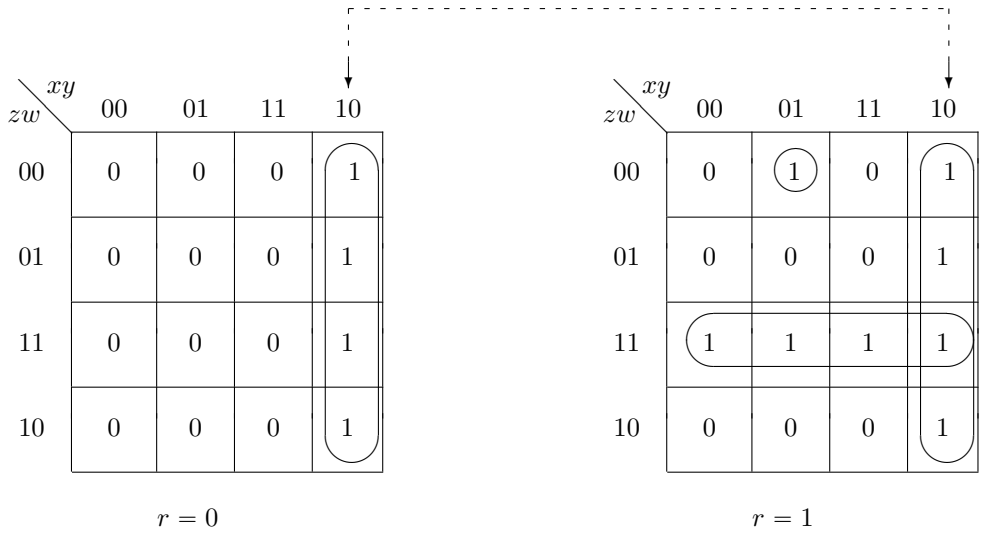


Figura 4.44: Uso della mappa di Karnaugh per la funzione (4.32) di 5 variabili

La funzione semplificata risulta

$$f = x\,\overline{y} + z\,w\,r + \overline{x}\,y\overline{z}\,\overline{w}\,r \tag{4.33}$$

Se le variabili fossero 6, p.es. x, y, z, w, r, t, sarebbe necessario usare quattro mappe nelle quattro variabili x, y, z, w, una per ogni coppia possibile di valori associati alle variabili r, t.

Le condizioni non specificate

Nella sintesi di una funzione logica di n variabili si può presentare il caso in cui per k configurazioni delle variabili di ingresso la funzione vale 1, per m configurazioni vale 0, ma $k + m < 2^n$. Le restanti $2^n - (m + k)$ configurazioni vengono dette *condizioni non specificate* (o anche *d.c.c.*, acronimo di *don't care condition*).
In pratica questa situazione si verifica ogni qual volta in un circuito certe configurazioni di ingresso sono fisicamente impossibili, o rendono privo di significato il valore dell'uscita. Da un punto di vista strettamente analitico ciò accade quando una funzione F è funzione delle variabili $\phi_1, \phi_2, \ldots, \phi_m$, ognuna delle quali è a sua volta funzione

delle variabili $x_1, x_2, ..., x_n$, cioè

$$F = f(\phi_1, \phi_2, \ldots, \phi_m) \qquad \phi_i = f_i(x_1, x_2, ..., x_n) \qquad i = 1, 2, ..., m$$

Le condizioni non specificate si ricavano dalle tavole di verità delle ϕ_i , e sono tutte e sole le configurazioni delle ϕ_i che non compaiono in tali tabelle. Sia ad esempio:

$$F = f(\phi_1, \phi_2, \phi_3) \qquad \begin{array}{l} \phi_1 = x\,(y\,z + \overline{y}\,\overline{z}) \\ \phi_2 = w\,(y\,\overline{z} + \overline{y}\,z) \\ \phi_3 = x\,y\,z + \overline{x}\,\overline{y}\,\overline{z} \end{array} \tag{4.34}$$

La tavola di verità delle ϕ_i è riportata nella figura 4.45. Nella seconda tavola della stessa figura sono riportati anche

x	y	z	w	ϕ_1	ϕ_2	ϕ_3
0	0	0	0	0	0	1
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	0
1	1	0	1	0	1	0
1	1	1	0	1	0	1
1	1	1	1	1	0	1

ϕ_1	ϕ_2	ϕ_3	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	–
1	0	0	1
1	0	1	0
1	1	0	–
1	1	1	–

Figura 4.45: Le tavole di verità delle funzioni ϕ_1, ϕ_2, ϕ_3, descritte dalle equazioni 4.34, e quella della funzione F

i valori assunti dalla funzione F; si vede che le terne di possibili valori ϕ_1, ϕ_2, ϕ_3 sono $000, 001, 010, 100, 101$, mentre le configurazioni rimanenti $011, 110, 111$ non compaiono. Ne consegue che la tavola di verità della F conterrà condizioni non specificate in corrispondenza di queste configurazioni d'ingresso.
Sulla tavola di verità le condizioni non specificate vengono indicate con un trattino, nella forma canonica raccogliendo in parentesi i termini minimi corrispondenti, sulle mappe di Karnaugh contrassegnando con il simbolo $\times$ la casella corrispondente a ciascuna condizione non specificata.
Le condizioni non specificate possono venir sfruttate nelle semplificazioni, in modo da pervenire ad espressioni minime più semplici. Se si opera con le mappe di Karnaugh, le semplificazioni vanno ancora fatte in modo da coprire tutte le caselle contrassegnate con un 1, ma se serve si possono aggregare anche le caselle associate a condizioni non specificate, che possiamo contrassegnare con $\times$; in pratica si tratta di utilizzare le condizioni non specificate per allargare al massimo i sottocubi di copertura della funzione, assegnando a ciascuna di loro il valore 1 o 0 a seconda che torni o meno utile per ottenere sottocubi più ampi. Nell'esempio di sopra si ottiene

ϕ_3 \ $\phi_1\phi_2$	00	01	11	10
0	1	1	×	1
1	0	×	×	0

Vediamo un altro esempio; si prenda la seguente funzione:

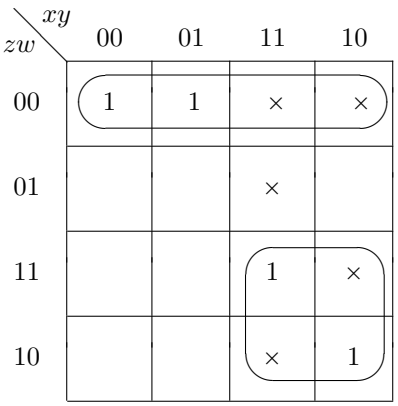


Figura 4.46: Semplificazione di una funzione usando le condizioni non specificate

Senza considerare le condizioni d.c.c. si otterrebbe:

$$F = \overline{x}\,\overline{z}\,\overline{w} + x\,y\,z\,w + x\,\overline{y}\,z\,\overline{w}$$

mentre tenendo conto anche di queste ultime si ha:

$$F = \overline{z}\,\overline{w} + x\,z$$

Il metodo tabellare di Quine - Mc Cluskey

Il metodo di *Quine - Mc Cluskey* è un procedimento tabellare che consente di ottenere la forma minima come somma di prodotti per qualsiasi funzione logica. Esso si basa sulla relazione:

$$f\,x + f\,\overline{x} = f$$

già usata in precedenza per le mappe di Karnaugh; solo che ora essa viene applicata in modo sistematico a tutti i termini minimi della funzione. Per capirne la logica riprendiamo la solita funzione di figura 4.20a:

		x	y	z	f	
0	$\overline{x}\overline{y}\overline{z}$	0	0	0	0	
1	$\overline{x}\overline{y}z$	0	0	1	0	
2	$\overline{x}y\overline{z}$	0	1	0	1	*
3	$\overline{x}yz$	0	1	1	1	*
4	$x\overline{y}\overline{z}$	1	0	0	0	
5	$x\overline{y}z$	1	0	1	1	*
6	$xy\overline{z}$	1	1	0	0	
7	xyz	1	1	1	1	*

Livello			x	y	z	f	
0	0	$\overline{x}\overline{y}\overline{z}$	0	0	0	0	
1	1	$\overline{x}\overline{y}z$	0	0	1	0	
	2	$\overline{x}y\overline{z}$	0	1	0	1	*
	4	$x\overline{y}\overline{z}$	1	0	0	0	
2	3	$\overline{x}yz$	0	1	1	1	*
	5	$x\overline{y}z$	1	0	1	1	*
	6	$xy\overline{z}$	1	1	0	0	
3	7	xyz	1	1	1	1	*

Livello			x	y	z	f	
1	2	$\overline{x}y\overline{z}$	0	1	0	1	*
	3	$\overline{x}yz$	0	1	1	1	*
2	5	$x\overline{y}z$	1	0	1	1	*
3	7	xyz	1	1	1	1	*

Figura 4.47: Trasformazione della tavola di verità della funzione di figura 4.20a per ottenere la tabella di Quine-Mc Cuskey

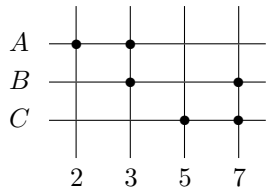
La semplificazione per via algebrica porta a

$$\overline{x}y\overline{z} + \overline{x}yz + x\overline{y}z + xyz = \overline{x}y(\overline{z} + z) + xz(\overline{y} + y) = \overline{x}y + xz \tag{4.35}$$

E' ben evidente che la struttura della semplificazione $\bar{x}y\bar{z} + \bar{x}yz = \bar{x}y(\bar{z} + z)$, che riguarda i termini minimi contrassegnati con $*$ in tabella, è del tipo $f\,\bar{z} + f\,z = f$ e riguarda due termini minimi che differiscono per la sola variabile z, che si trova diretta in uno dei due termini e negata nell'altro. La stessa cosa si può dire per i due termini minimi contrassegnati con $*$. Ordiniamo allora le righe della tabella in modo da mettere su livelli adiacenti le n-ple che differiscono per una sola variabile. Una possibilità è quella di procedere per peso crescente delle n-ple binarie associate ai termini minimi - dove per peso di un n-pla s'intende il numero di 1 in essa presenti - in modo che a ciascun peso corrisponda un livello; così facendo si ottiene la seconda tabella di figura 4.47. Se ora eliminiamo le righe che corrispondono a termini minimi per i quali la funzione vale 0 - che non rientrano nella somma di prodotti - otteniamo la terza tabella di figura 4.47, che corrisponde alla tabella di Quine-Mc Cuskey. A questo punto si può procedere con le semplificazioni, che generano una seconda tabella. Poichè $\bar{x}y\bar{z}$ si semplifica con $\bar{x}yz$ per ottenere $\bar{x}y$, su questa tabella segniamo con una lineetta la variabile z che è stata oggetto di semplificazione; segniamo inoltre a lato i termini minimi che sono coinvolti nella semplificazione, cioè $2-3$; viceversa la coppia $2-5$ non porta ad alcuna semplificazione. Esaurito il confronto tra i livelli 1 e 2, possiamo ora controllare le semplificazioni tra i livelli 2 e 3; in questo caso troviamo semplificazioni per entrambe le coppie, cioè $3-7$ e $5-7$. La tabella che si ottiene è la seguente:

		x	y	z	
	$2-3$	0	1	$-$	A
	$3-7$	$-$	1	1	B
	$5-7$	1	$-$	1	C

Poichè non è possibile fare altre semplificazioni ci si ferma, contrassegnando con delle lettere progressive gli implicanti che non si possono più semplificare. L'espressione finale deriva dalla somma degli implicanti coinvolti, cioè $f = A + B + C = \bar{x}y + yz + xz$. Si osservi tuttavia che l'espressione che otteniamo non è in forma minima, poiché come noto il termine yz si può eliminare per il terzo teorema dell'assorbimento. Per individuare il minimo numero di implicanti che implicano tutti i termini minimi dobbiamo costruire il seguente reticolo, disponendo sulle righe gli implicanti, sulle colonne i termini minimi, e ponendo un segno (p.es. un pallino) in corrispondenza dei termini minimi che formano ciascun implicante.



Procedendo per colonne osserviamo che il termine minimo 2 è coperto solo da A e dunque questo implicante è indispensabile; il termine minimo 5 è coperto solo da C e dunque anche questo implicante è indispensabile. L'implicante B non è invece necessario alla composizione della funzione, poiché i termini minimi a lui associati, 3 e 7, sono già coperti rispettivamente da A e da C. Dunque la funzione semplificata è $f = A + C = \bar{x}y + xz$.

La procedura completa per la semplificazione mediante il metodo tabellare di Quine-Mc Cluskey è dunque la seguente:

1. si esprimono i termini minimi che sono a 1 sostituendo ogni variabile diretta con 1 e ogni variabile negata con 0. Ad esempio il termine minimo $\bar{x}yz$ viene rappresentato con 011;
2. si ordinano tutti i termini minimi in gruppi aventi lo stesso peso, cioè lo stesso numero di 1; tali raggruppamenti vengono chiamati livelli;
3. si costruisce una tabella disponendo i livelli in ordine crescente e associando a ciascun termine minimo il corrispondente numero decimale. Ad esempio i termini minimi

$$\bar{x}\,\bar{y}\,\bar{z} \qquad \bar{x}\,\bar{y}\,z \qquad \bar{x}\,y\,\bar{z} \qquad \bar{x}\,y\,z \qquad x\,y\,z$$

generano la seguente tabella

	Livello	Numero	Termine minimo
$\surd$	0	0	000
$\surd$	1	1	001
$\surd$		2	010
$\surd$	2	3	011
$\surd$	3	7	111

4. a partire dal primo termine minimo del primo livello disponibile, si confrontano tutti i termini minimi del livello k con tutti quelli del livello $k+1$, semplificando tra loro i termini che differiscono per un solo bit. Si costruisce in tal modo una seconda tabella, nella quale le semplificazioni avvenute si indicano con una lineetta, mentre gli implicanti vengono contraddistinti con i numeri dei termini minimi che li hanno generati. Nella tabella si contrassegnano tutti i termini minimi che hanno dato luogo ad almeno una semplificazione, mentre i termini minimi che non hanno portato a semplificazioni vengono contrassegnati con lettere progressive dell'alfabeto. Riferendosi all'esempio riportato sopra, si ottiene la seguente tabella

$\surd$	$0,1$	$0\ 0-$
$\surd$	$0,2$	$0-0$
$\surd$	$1,3$	$0-1$
$\surd$	$2,3$	$0\ 1-$
A	$3,7$	$-1\ 1$

5. con lo stesso procedimento di prima, nella seconda tabella si confrontano tutti i termini del livello k con tutti quelli del livello $k+1$. Sono semplificabili tra loro i termini che differiscono per un solo bit e che siano già stati semplificati rispetto alla stessa variabile. Si costruisce in tal modo una terza tabella con le stesse modalità esposte per la costruzione della seconda tabella. Nell'esempio che si sta trattando si ha

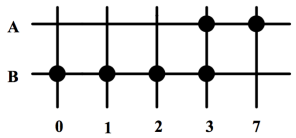
$$B \mid 0,1,2,3 \mid \quad 0\ -\ -$$

mentre nella seconda tabella il termine $3,7$ non dà luogo a semplificazioni e viene quindi contrassegnato con A.

6. si prosegue in modo analogo, con la costruzione di tabelle successive, finché non è più possibile eseguire semplificazioni. Tutti i termini che nelle successive tabelle non sono stati contrassegnati vengono chiamati *implicanti primi* e la loro somma logica realizza la funzione desiderata. Può però accadere che l'espressione minima di tale funzione si realizza però con un numero di implicanti inferiore a quello degli implicanti primi. Essa infatti si ricava come somma del minimo numero di implicanti primi con cui vengono implicati tutti i termini minimi della funzione. Nell'esempio che si sta esaminando gli implicanti primi sono

$$A = -\,1\,1 \qquad B = 0\ -\ -$$

7. la scelta più opportuna degli implicanti primi necessari, che può divenire complessa già con un numero di variabili relativamente ridotto, avviene mediante la costruzione di un semplice *reticolo*, avente i termini minimi sulle colonne e gli implicanti primi sulle righe. Su ogni riga, cioè in corrispondenza di ciascun implicante, si contrassegnano opportunamente i termini minimi implicati. Nell'esempio trattato si ottiene il seguente reticolo



Dall'esame del reticolo si individuano poi i termini minimi che sono implicati da un unico implicante primo. Ciascuno di essi diviene evidentemente essenziale nella realizzazione della funzione. Ogni implicante essenziale così individuato implica d'altra parte altri termini minimi che risultano automaticamente coperti e pertanto non vanno più considerati. E' semplice infine trovare, sia pure per tentativi, la copertura minima dei termini rimasti. Nell'esempio che si sta trattando ambedue gli implicanti primi A e B sono essenziali; l'espressione semplificata è allora

$$f = A + B = y\,z + \overline{x}$$

Esempio 4.4. Più significativo, soprattutto per quanto riguarda la realizzazione della copertura minima, è l'esempio seguente. Si voglia semplificare la funzione

$$f = \sum(1, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14, 15)_m \tag{4.36}$$

nella quale la notazione binaria associata a ciascun numero intero rappresenta la codifica dei termini minimi. La divisione dei termini minimi in livelli e il riordinamento dei livelli dà luogo alla tabella di sinistra, mentre al centro e sulla destra ci sono le successive tabelle di semplificazione

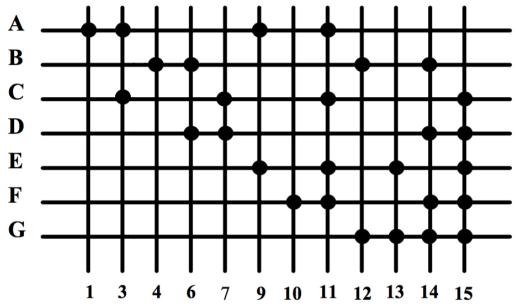
	Livello	Numero	Termine minimo
√	1	1	0001
√	1	4	0100
√	2	3	0011
√		6	0110
√	2	9	1001
√	2	10	1010
√	2	12	1100
√	3	7	0111
√	3	11	1011
√	3	13	1101
√	3	14	1110
√	4	15	1111

√	1, 3	0 0 − 1
√	1, 9	− 0 0 1
√	4, 6	0 1 − 0
√	4, 12	− 1 0 0
√	3, 7	0 − 1 1
√	3, 11	− 0 1 1
√	6, 7	0 1 1 −
√	6, 14	− 1 1 0
√	9, 11	1 0 − 1
√	9, 13	1 − 0 1
√	10, 11	1 0 1 −
√	10, 14	1 − 1 0
√	12, 13	1 1 0 −
√	12, 14	1 1 − 0
√	7, 15	− 1 1 1
√	11, 15	1 − 1 1
√	13, 15	1 1 − 1
√	14, 15	1 1 1 −

A	1, 3, 9, 11	− 0 − 1
B	4, 6, 12, 14	− 1 − 0
C	3, 7, 11, 15	− − 1 1
D	6, 7, 14, 15	− 1 1 −
E	9, 11, 13, 15	1 − − 1
F	10, 11, 14, 15	1 − 1 −
G	12, 13, 14, 15	1 1 − −

Il reticolo di semplificazione è invece rappresentato in figura 4.4.

Dall'esame del reticolo risulta che il termine minimo 1 è implicato solamente da A, quello 4 solo da B e quello 10 solo da F. Di conseguenza A, B, F sono implicanti primi essenziali. La loro scelta copre i termini minimi $1, 3, 4, 6, 9, 10, 11, 12, 14, 15$. Per coprire i rimanenti termini minimi 7 e 13 si può scegliere per il primo l'implicante C o quello D, per il secondo quello E o quello G.



Ci sono pertanto quattro realizzazioni equivalenti della funzione assegnata

$$
\begin{aligned}
f &= A + B + F + D + E = \overline{y}\, w + y\, \overline{w} + x\, z + y\, z + x\, w \\
f &= A + B + F + D + G = \overline{y}\, w + y\, \overline{w} + x\, z + y\, z + x\, y \\
f &= A + B + F + C + E = \overline{y}\, w + y\, \overline{w} + x\, z + z\, w + x\, w \\
f &= A + B + F + C + G = \overline{y}\, w + y\, \overline{w} + x\, z + z\, w + x\, y
\end{aligned}
\tag{4.37}
$$

Tali realizzazioni si comprendono non appena si valuti attentamente la mappa di Karnaugh 4.48 associata alla funzione 4.36

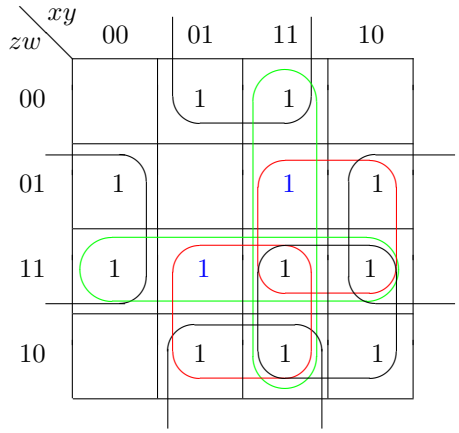


Figura 4.48: Mappa di Karnaugh

Si può vedere che i sottocubi neri sono necessari per coprire i corrispondenti termini minimi, e sono i primi tre termini della 4.37; per coprire però i termini minimi associati ai due 1 color blu ci sono i due sottocubi rossi oppure i due sottocubi verdi; poiché sono tra loro indipendenti, possiamo completare la copertura con entrambi i sottocubi rossi, entrambi i sottocubi verdi, oppure uno rosso e uno verde nei due modi possibili; queste combinazioni sono quelle relative agli ultimi due termini della 4.37.

Capitolo 5

Circuiti combinatori

5.1 Introduzione

Si chiamano *combinatori* quei circuiti dotati di uno o più ingressi e uno o più uscite, il cui funzionamento è descritto da una funzione logica; in questi circuiti gli ingressi e le uscite possono assumere solo uno di due valori binari previsti (0 e 1); inoltre in ogni istante l'uscita è funzione deterministica unicamente degli ingressi. La figura 5.1 ci illustra la struttura generale di un circuito combinatorio con n ingressi e m uscite, il cui funzionamento è

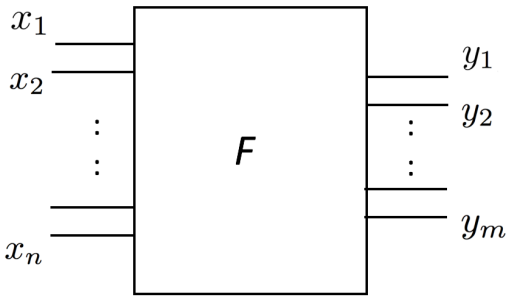


Figura 5.1: Struttura generale di un circuito combinatorio

descritto dalla funzione $F : \mathbf{2^n} \rightarrow \mathbf{2^m}$, che è una funzione Booleana da $\mathbf{2^n}$ a $\mathbf{2^m}$ realizzata mediante m funzioni Booleane $f_i : \mathbf{2^n} \rightarrow \mathbf{2}$

$$\begin{aligned} y_1 &= f_1(x_1, x_2, \ldots, x_n) \\ y_2 &= f_2(x_1, x_2, \ldots, x_n) \\ &\vdots \\ y_m &= f_m(x_1, x_2, \ldots, x_n) \end{aligned}$$

Nei circuiti elettronici i due stati 0 e 1 sono realizzati mediante due livelli caratteristici di tensione, detti livello alto (*h*) e livello basso (*l*). L'effettiva corrispondenza tra *h* e *l* e le costanti logiche 0 e 1 è convenzionale e va precisata di volta in volta.
E' detta *logica positiva* la convenzione secondo la quale il valore 1 viene associato al livello alto *h*; *logica negativa*

quello in cui il valore 1 è associato al livello basso l.
Si chiama *circuito logico elementare* o *porta logica* un circuito a 1 o 2 ingressi e un'uscita il cui valore è 1 in corrispondenza delle configurazioni degli ingressi descritte dalle funzioni logiche NOT (per 1 ingresso), OR, AND, NAND, NOR, XOR, XNOR (per 2 ingressi). Indipendentemente dalla loro realizzazione circuitale e dal tipo di logica (positiva o negativa), le porte logiche vengono indicate graficamente con i simboli che abbiamo già incontrato e che illustriamo nuovamente in figura 5.2.

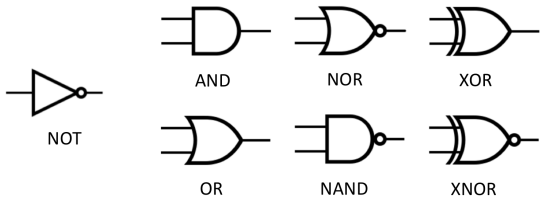


Figura 5.2: Simboli grafici delle principali porte logiche a uno e due ingressi

5.1.1 Itinerari e livelli

Ogni rete logica è formata da un certo numero di porte (AND, OR, ecc.) tra loro variamente interconnesse, da un certo numero di ingressi, contraddistinti in figura 5.3 con i simboli A_i, e da un certo numero di uscite, contraddistinte nella medesima figura con i simboli B_k. Si dice *itinerario* tra due elementi X e Y qualsiasi

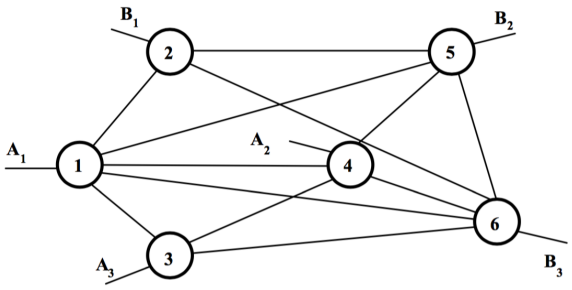
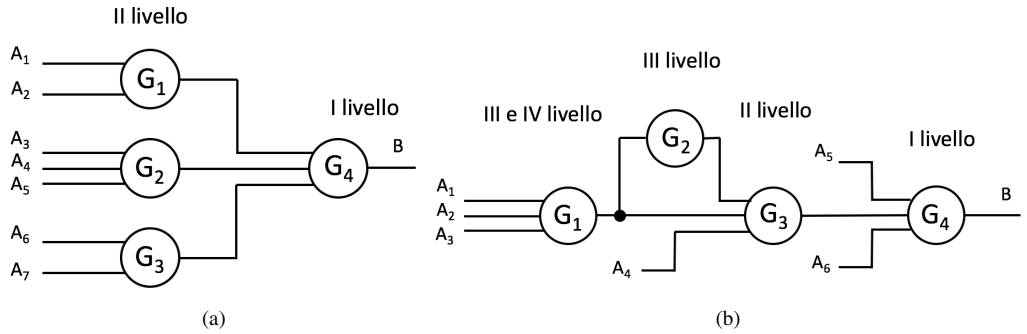


Figura 5.3: Schematizzazione di una rete logica complessa

percorso che colleghi X con Y. Si dice invece *livello di un elemento* X rispetto all'uscita B_j e a un determinato itinerario I, il numero di elementi, X compreso, disposti lungo l'itinerario a partire dall'uscita B_j. Si dice *livello di una variabile* rispetto all'uscita B_j e all'itinerario I il numero di elementi compresi tra il rispettivo ingresso e l'uscita B_j lungo l'itinerario I. In figura 5.4 sono riportati due esempi.



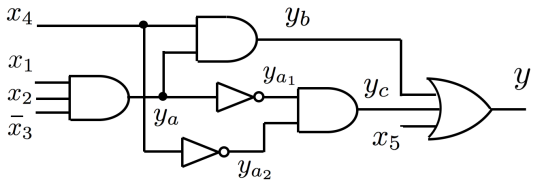
(a) (b)

Figura 5.4: Esempi di livelli in due reti logiche

Si noti che uno stesso elemento può avere più livelli a secondo dell'itinerario scelto. Ad esempio in figura 5.4b il gate G1 e' di III livello secondo l'itinerario G4, G3, G1 e di IV livello secondo l'itinerario G4, G3, G2, G1.
Per i circuiti combinatori, come per ogni altro tipo di circuito, si pongono due problemi opposti: da un lato quello dell'*analisi*, cioè della descrizione del funzionamento del circuito, una volta che sia nota la sua configurazione, il che significa la definizione della funzione che realizza; dall'altro quello della *sintesi*, cioè del progetto di un circuito che realizzi una certa funzione logica, comunque descritta.

5.2 Analisi dei circuiti combinatori

L'analisi di un circuito combinatorio tende a ottenere una rappresentazione della funzione d'uscita y o nella sua forma analitica, oppure sotto forma di tavola di verità. Poiché la rappresentazione circuitale è simbolica, l'analisi non è legata a considerazioni di logica positiva o negativa.
Per effettuare l'analisi, nel caso di circuiti AND-OR-NOT, è sufficiente partire dagli elementi su cui entrano le variabili e procedere verso il terminale di uscita secondo tutti i possibili itinerari, usando la funzione di uscita di ciascun elemento come variabile di ingresso dell'elemento successivo. Nella figura 5.5 vediamo un esempio di una rete e delle equazioni a essa associate, ricavate col procedimento sopra descritto. L'analisi dei circuiti basati



(a) Rete AND-OR-NOT da analizzare

$$y_a = x_1\, x_2\, \overline{x_3}$$
$$y_{a_1} = \overline{y_a} = \overline{x_1\, x_2\, \overline{x_3}} = \overline{x_1} + \overline{x_2} + x_3$$
$$y_{a_2} = \overline{x_4}$$
$$y_b = x_4\, y_a = x_1\, x_2\, \overline{x_3}\, x_4$$
$$y_c = y_{a_1}\, y_{a_2} = \overline{x_4}\, (\overline{x_1} + \overline{x_2} + x_3)$$
$$y = y_b + y_c + x_5 = x_1\, x_2\, \overline{x_3}\, x_4 + \overline{x_4}\, (\overline{x_1} + \overline{x_2} + \overline{x_3}) + x_5$$

(b) Equazioni dell'analisi

Figura 5.5: Rete AND-OR-NOT ed equazioni che si ottengono dall'analisi

su porte NAND e NOR è notevolemente più complessa, a causa della non associatività degli operatori. Lasciamo questo tipo di analisi al corso di *Reti Logiche*.

5.3 Sintesi dei circuiti combinatori

Eseguire la sintesi di un circuito combinatorio consiste, come già accennato, nel progettare un circuito a *n* ingressi che soddisfi una determinata funzione di uscita y di progetto. La funzione y che il circuito deve soddisfare

può essere assegnata in diverse forme. Più precisamente:

1. con la descrizione a parole del funzionamento del circuito. E' questa la forma di assegnazione più comune, ma anche la più imprecisa. E' necessario porre un'estrema attenzione alla corretta interpretazione di eventuali condizioni implicite e all'esistenza di vincoli di qualsiasi natura. Dalla descrizione verbale è necessario passare poi alla tavola di verità, assegnando il valore della funzione per ognuna delle 2^n configurazioni degli ingressi;
2. con una vera e propria tavola di verità, che è in definitiva l'effettivo punto di partenza della sintesi cui tutti gli altri tipi di assegnazione devono essere ricondotti;
3. con un'espressione analitica, che è il modo più conciso, anche se non univoco, di descrivere il funzionamento di un circuito;
4. con uno schema logico; è questa una procedura generalmente usata quando un determinato circuito logico debba esser riprogettato con componenti diversi. In tal caso, con le regole dell'analisi si ricava un'espressione analitica della funzione y.

Qualunque sia il metodo di assegnazione, la sintesi procede partendo dalla tavola di verità o da un'espressione analitica; applicando i metodi di semplificazione delle funzioni logiche si giunge alla forma più conveniente per gli scopi che ci si propone. Si noti che non sempre la forma più conveniente corrisponde alla forma minima della funzione. Ad esempio non è sempre opportuno realizzare circuitalmente la forma minima algebrica, in quanto vi possono essere dei vincoli sul numero massimo di livelli. Infatti, se è ben vero quanto esposto precedentemente, e cioè che nei circuiti combinatori l'uscita è, istante per istante, funzione unicamente degli ingressi, non significa che la variazione degli ingressi sia avvertita immediatamente in uscita; tale affermazione sta piuttosto a significare che ogni configurazione di ingresso dà luogo a una determinata uscita e che eventuali transitori di commutazione possono ritardare, ma non modificare quest'uscita.
Poiché il tempo di commutazione Δ di qualsiasi porta logica, per quanto piccolo, non è mai nullo, il tempo di risposta di un circuito a n livelli, al variare della configurazione d'ingresso, è $n \cdot \Delta$. In definitiva, il ritardo totale tra ingresso e uscita è proporzionale al numero di livelli e potendo la forma minima di una funzione contenere un numero di livelli molto elevato, la sua diretta realizzazione circuitale potrebbe dar luogo a ritardi intollerabili.
La forma in cui si ha il minimo ritardo è quella a due livelli, che d'altra parte è quella che si ottiene con i metodi di semplificazione che sono stati esposti nel capitolo 4. La convenienza di eventuali fattorizzazioni va valutata caso per caso.
Si può dunque concludere che la sintesi di un circuito combinatorio procede attraverso i seguenti passi:

1. Descrizione del funzionamento del circuito
2. Determinazione della tavola di verità
3. Sintesi della funzione Booleana
4. Semplificazione della funzione logica relativa
5. Determinazione della forma minima più conveniente
6. Disegno del circuito

Si osservi che il passo (5) non può essere attuato secondo un procedimento sistematico, e l'effettiva forma minima più conveniente andrà valutata di volta in volta, eventualmente facendo uso di tecniche basate sul concetto di decomponibilità che saranno illustrate nel successivo corso di *Reti Logiche*.

Esempio 5.1. Si voglia realizzare un circuito a tre ingressi e quattro uscite; sugli ingressi si può presentare un numero binario compreso tra 0 e 5. All'uscita di tale circuito si deve ottenere il prodotto per 3 del numero in ingresso. Il massimo numero di uscita rappresentabile con 4 bit è 15 e sarà necessario sintetizzare quattro funzioni logiche.

	$\boldsymbol{x}$			$\boldsymbol{y}$			
$3 \cdot \boldsymbol{x} = \boldsymbol{y}$	x_2	x_1	x_0	y_3	y_2	y_1	y_0
$3 \cdot 0 = 0$	0	0	0	0	0	0	0
$3 \cdot 1 = 3$	0	0	1	0	0	1	1
$3 \cdot 2 = 6$	0	1	0	0	1	1	0
$3 \cdot 3 = 9$	0	1	1	1	0	0	1
$3 \cdot 4 = 12$	1	0	0	1	1	0	0
$3 \cdot 5 = 15$	1	0	1	1	1	1	1
$3 \cdot 6 = -$	1	1	0	$-$	$-$	$-$	$-$
$3 \cdot 7 = -$	1	1	1	$-$	$-$	$-$	$-$

Figura 5.6: Tavola di verità del circuito che moltiplica per 3.

Le tavole di verità di ogni funzione sono riportate in figura 5.6, mentre in figura 5.7 si hanno le corrispondenti mappe di Karnaugh.

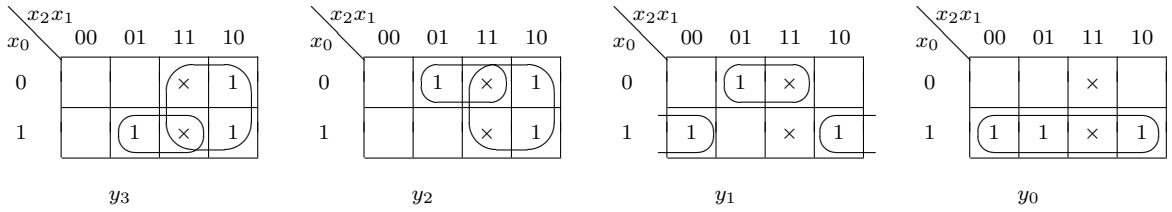


Figura 5.7: Mappe di Karnaugh per le quattro funzioni di figura 5.6

Utilizzando opportunamente le condizioni non specificate si ottiene:

$$y_3 = x_2 + x_0\, x_1 \qquad y_1 = \overline{x_0}\, x_1 + x_0\, \overline{x_1}$$
$$y_2 = x_2 + \overline{x_0}\, x_1 \qquad y_0 = x_0$$

cui corrisponde il circuito di figura 5.8, ottenuto mettendo in comune il termine $\overline{x_0}x_1$ tra le funzioni y_1 e y_2. ◯

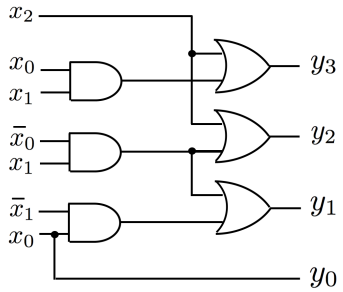
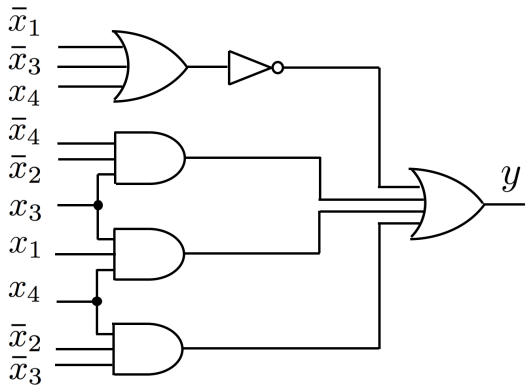
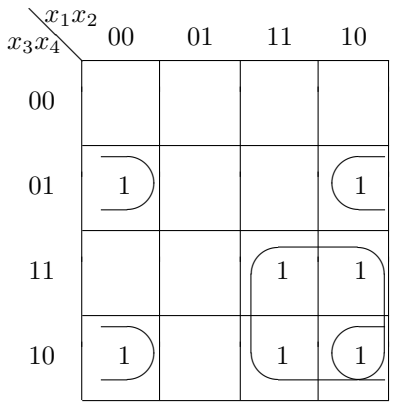


Figura 5.8: Moltiplicatore per 3 di un numero compreso tra 0 e 5



(a) Circuito assegnato da semplificare



(b) Mappa di Karnaugh della funzione 5.1

Figura 5.9: Semplificazione di un circuito mediante mappa di Karnaugh

Esempio 5.2. Si voglia sintetizzare un circuito con la stessa funzione Booleana di quello illustrato in figura 5.9a, ma possibilmente più economico.

L'espressione analitica della funzione di trasmissione è

$$\begin{aligned} y &= \overline{\overline{x_1} + \overline{x_3} + x_4} + \overline{x_2}\, x_3\, \overline{x_4} + x_1\, x_3\, x_4 + \overline{x_2}\, \overline{x_3}\, x_4 = \\ &= x_1\, x_3\, \overline{x_4} + \overline{x_2}\, x_3\, \overline{x_4} + x_1\, x_3\, x_4 + \overline{x_2}\, \overline{x_3}\, x_4 \end{aligned} \tag{5.1}$$

Dalla mappa di Karnaugh di figura 5.9b si ricava l'espressione minima a due livelli

$$y = x_1\, x_3 + \overline{x_2}\, \overline{x_3}\, x_4 + \overline{x_2}\, x_3\, \overline{x_4} \tag{5.2}$$

e fattorizzando si ottiene:

$$y = x_1\, x_3 + \overline{x_2}\, (\overline{x_3}\, x_4 + x_3\, \overline{x_4}) \tag{5.3}$$

Per la realizzazione della (5.2) sono sufficienti 4 porte, mentre per la (5.3) ne sono necessarie 6. Il circuito che

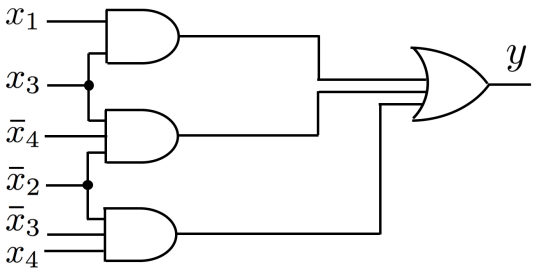


Figura 5.10: Circuito più economico che realizza la stessa funzione Booleana del circuito di figura 5.9a

conviene realizzare è allora quello relativo all'espressione (5.2), ed è riportato in figura 5.10. ◯

Si noti che negli esempi fatti si è supposto di avere a disposizione all'ingresso del circuito sia le variabili dirette che la loro negazione. Quando questa situazione non si verifica anche il numero di invertitori va minimizzato. Ad esempio la funzione

$$y = \overline{x_1} + \overline{x_2} + \overline{x_3} + \overline{x_4}$$

realizzabile con quattro invertitori e un gate OR, può essere realizzata molto più convenientemente tenendo presente che

$$y = \overline{x_1} + \overline{x_2} + \overline{x_3} + \overline{x_4} = \overline{x_1\, x_2\, x_3\, x_4}$$

richiedendo in tal caso solamente un AND e un NOT.
Facciamo un altro esempio di gran lunga più impegnativo dei precedenti, tratto da [3].

Esempio 5.3. Si voglia costruire un circuito che esegue la moltiplicazione tra due numeri espressi in notazione posizionale in base 2. Per mantenere una complessità gestibile in un contesto didattico, limitiamoci a due numeri da due bit, e dunque un moltiplicatore binario a 2×2 bit. Siano $\boldsymbol{x} = x_1x_0$ i bit del moltiplicando e $\boldsymbol{y} = y_1y_0$ i bit del moltiplicatore; ciascun fattore varia dunque nell'intervallo $[0, 3]$, mentre il prodotto è compreso nell'intervallo $[0, 9]$. Il progetto richiede la costruzione della tavola di verità per tutte le possibili 16 combinazioni, tenendo conto che per rappresentare il valore del prodotto serviranno $\lceil \log_2 9 \rceil = 4$ bit. Se, per esempio, si ha $\boldsymbol{x} = 10\ (2)$ e $\boldsymbol{y} = 11\ (3)$, a questa configurazione d'ingresso deve corrispondere $\boldsymbol{z} = 0110\ (6)$, considerato che $2 \cdot 3 = 6$. Operando in questo modo si ottiene la tavola di verità di figura 5.11, nella quale ogni colonna i-esima rappresenta i valori assunti dalla funzione z_i in corrispondenza delle varie quaterne d'ingresso. Dobbiamo allora valutare quattro

	$\boldsymbol{x}$		$\boldsymbol{y}$		$\boldsymbol{z}$			
$\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{z}$	x_1	x_0	y_1	y_0	z_3	z_2	z_1	z_0
$0 \cdot 0 = 0$	0	0	0	0	0	0	0	0
$0 \cdot 1 = 0$	0	0	0	1	0	0	0	0
$0 \cdot 2 = 0$	0	0	1	0	0	0	0	0
$0 \cdot 3 = 0$	0	0	1	1	0	0	0	0
$1 \cdot 0 = 0$	0	1	0	0	0	0	0	0
$1 \cdot 1 = 1$	0	1	0	1	0	0	0	1
$1 \cdot 2 = 2$	0	1	1	0	0	0	1	0
$1 \cdot 3 = 3$	0	1	1	1	0	0	1	1
$2 \cdot 0 = 0$	1	0	0	0	0	0	0	0
$2 \cdot 1 = 2$	1	0	0	1	0	0	1	0
$2 \cdot 2 = 4$	1	0	1	0	0	1	0	0
$2 \cdot 3 = 6$	1	0	1	1	0	1	1	0
$3 \cdot 0 = 0$	1	1	0	0	0	0	0	0
$3 \cdot 1 = 3$	1	1	0	1	0	0	1	1
$3 \cdot 2 = 6$	1	1	1	0	0	1	1	0
$3 \cdot 3 = 9$	1	1	1	1	1	0	0	1

Figura 5.11: Tavola di verità per il moltiplicatore binario a 2×2 bit

funzioni Booleane del tipo $\mathbf{2^2} \rightarrow \mathbf{2}$. Poichè ci sono pochi 1 e molti 0, conviene usare la I forma canonica, basata

sui termini minimi. Facendo i calcoli si ottiene

$$
\begin{aligned}
z_0 &= \overline{x_1}\, x_0\, \overline{y_1}\, y_0 + \overline{x_1}\, x_0\, y_1\, y_0 + x_1\, x_0\, \overline{y_1}\, y_0 + x_1\, x_0\, y_1\, y_0 \\
&= \overline{x_1}\, x_0\, y_0(\overline{y_1} + y_1) + x_1\, x_0\, y_0(\overline{y_1} + y_1) \\
&= \overline{x_1}\, x_0\, y_0 + x_1\, x_0\, y_0 \\
&= x_0\, y_0(\overline{x_1} + x_1) \\
&= x_0\, y_0
\end{aligned}
$$

$$
\begin{aligned}
z_1 &= \overline{x_1}\, x_0\, y_1\, \overline{y_0} + \overline{x_1}\, x_0\, y_1\, y_0 + x_1\, \overline{x_0}\, \overline{y_1}\, y_0 + x_1\, \overline{x_0}\, y_1\, y_0 + x_1\, x_0\, \overline{y_1}\, y_0 + x_1\, x_0\, y_1\, \overline{y_0} \\
&= \overline{x_1}\, x_0\, y_1(\overline{y_0} + y_0) + x_1\, \overline{x_0}\, y_0(\overline{y_1} + y_1) + x_1\, x_0\, \overline{y_1}\, y_0 + x_1\, x_0\, y_1\, \overline{y_0} \\
&= \overline{x_1}\, x_0\, y_1 + x_1\, \overline{x_0}\, y_0 + x_1\, x_0\, \overline{y_1}\, y_0 + x_1\, x_0\, y_1\, \overline{y_0} \\
&= x_0\, y_1(\overline{x_1} + x_1\, \overline{y_0}) + x_1\, y_0(\overline{x_0} + x_0\, \overline{y_1}) \\
&= x_0\, y_1(\overline{x_1} + \overline{y_0}) + x_1\, y_0(\overline{x_0} + \overline{y_1}) \\
&= \overline{x_1}\, x_0\, y_1 + x_0\, y_1\, \overline{y_0} + x_1\, \overline{x_0}\, y_0 + x_1\, \overline{y_1}\, y_0
\end{aligned}
$$

$$
\begin{aligned}
z_2 &= x_1\, \overline{x_0}\, y_1\, \overline{y_0} + x_1\, \overline{x_0}\, y_1\, y_0 + x_1\, x_0\, y_1\, \overline{y_0} \\
&= x_1\, \overline{x_0}\, y_1(\overline{y_0} + y_0) + x_1\, x_0\, y_1\, \overline{y_0} \\
&= x_1\, y_1(\overline{x_0} + x_0\overline{y_0}) \\
&= x_1\, y_1(\overline{x_0} + \overline{y_0}) \\
&= x_1\overline{x_0}\, y_1 + x_1\, y_1\, \overline{y_0}
\end{aligned}
$$

$$
z_3 = x_1\, x_0\, y_1\, y_0
$$

Risulta però molto più agevole ricavare gli implicanti direttamente dalle mappe di Karnaugh delle quattro funzioni z_3, z_2, z_1, z_0, come rappresentato in figura 5.12.

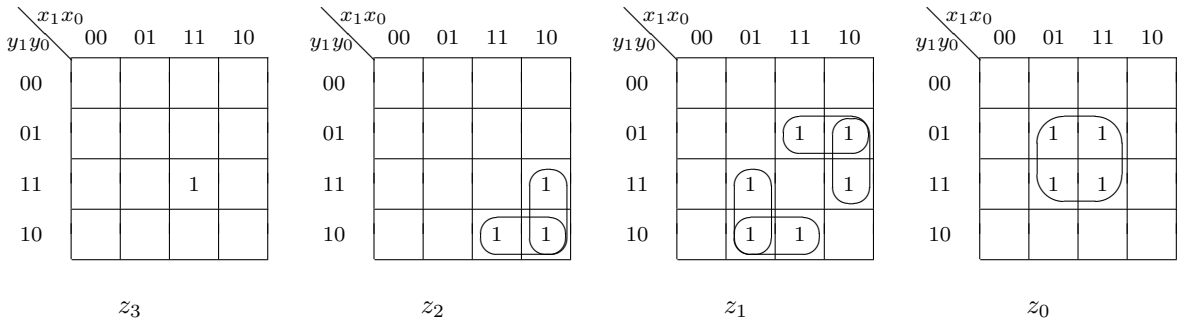


Figura 5.12: Mappe di Karnaugh per le quattro funzioni di figura 5.11

Il circuito completo del moltiplicatore binario è rappresentato in figura 5.13; esso è ricavato direttamente dalle espressioni di z_i usando porte AND, OR e NOT.

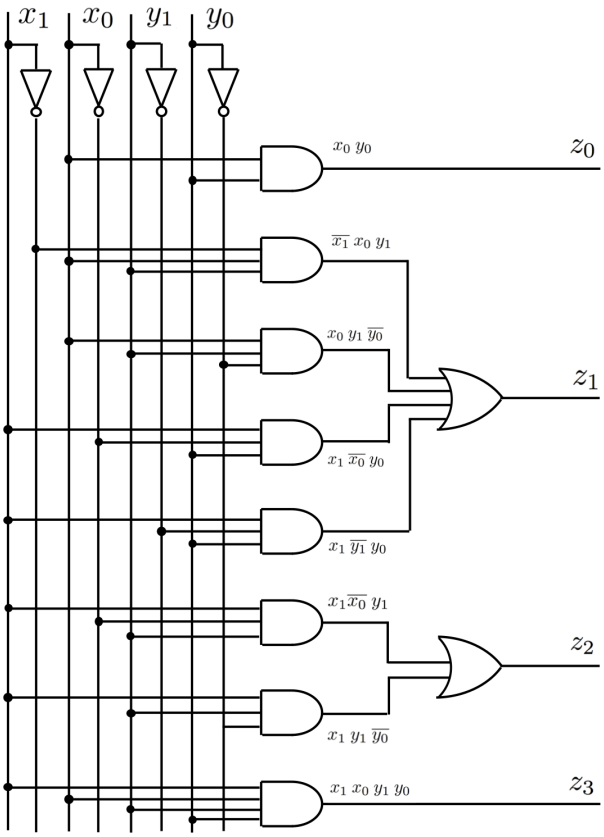


Figura 5.13: Moltiplicatore binario a 2×2 bit

5.4 Moduli combinatori

L'algebra Booleana permette di analizzare e progettare qualunque tipo di rete combinatoria, di qualunque complessità. In precedenza abbiamo mostrato anche le tecniche per semplificare le reti ottenute direttamente dalle espressioni delle forme canoniche, basate sulla somma di prodotti o sui prodotti di somme. Sebbene in linea di principio quei metodi possono essere utilizzati per reti combinatorie di qualunque estensione, nella pratica si preferisce affrontare il progetto di una rete complessa secondo una tecnica di scomposizione della rete in blocchi funzionali, detti *moduli combinatori*. In questo modo si rinuncia alla soluzione teoricamente ottima, a favore di una maggiore comprensibilità, gestibilità e soprattutto modularità del progetto. Ciò porta anche a un effettivo risparmio economico, poiché i moduli combinatori, se prodotti in larga scala, hanno prezzi via via sempre più bassi; dunque non sempre un numero minore di porte implica un risparmio economico.
Già a partire dagli anni '70 vennero prodotti moduli integrati, che svolgendo precise funzionalità di carattere combinatorio trovarono impiego nello sviluppo di progetti complessi. Nella parte che segue vengono esaminati alcuni moduli combinatori di uso corrente.

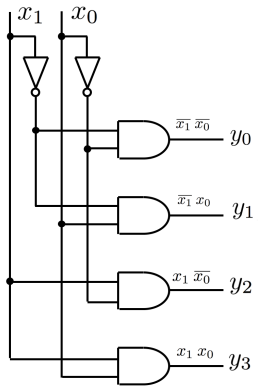
5.4.1 Decodificatori

I decodificatori convertono un numero espresso in notazione posizionale in base 2 in un numero intero in base 10. Un decodificatore accetta in ingresso n bit e presenta in uscita $m = 2^n$ linee, numerate da 0 a $2^n - 1$, in modo tale che va a 1 la sola linea y_j che corrisponde all'intero j codificato in notazione posizionale dagli n bit d'ingresso. Se dunque indichiamo con y_0, y_1, ..., y_m le uscite del decodificatore, la generica uscita y_j è ottenuta come AND delle n variabili che compongono il *minterm* j. Nelle figure 5.14a e 5.14b vengono riportati, rispettivamente, la tavola di verità e il circuito di un decodificatore a 2 bit; analogamente, nelle figure 5.15a e 5.15b abbiamo la tavola di verità e il circuito di un decodificatore a 3 bit. In figura 5.14c viene invece rappresentato il simbolo circuitale per un generico decodificatore a n bit.

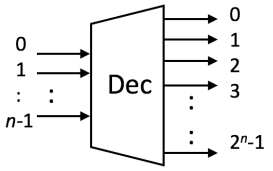
$\boldsymbol{x}$		$\boldsymbol{y}$			
x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$y_0 = \overline{x_1}\ \overline{x_0}$$
$$y_1 = \overline{x_1}\ x_0$$
$$y_2 = x_1\ \overline{x_0}$$
$$y_3 = x_1\ x_0$$

(a) Tavola di verità di un decodificatore a 2 bit



(b) Circuito del decodificatore a 2 bit



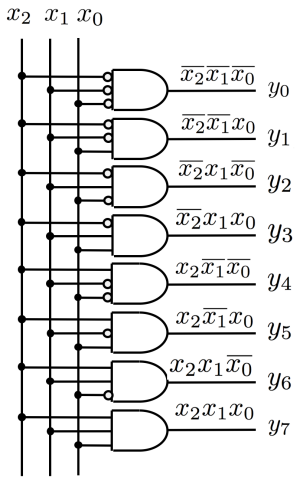
(c) Simbolo circuitale di un decodificatore a n bit

Figura 5.14: Decodificatore a 2 bit e simbolo circuitale di un codificatore a n bit

$\boldsymbol{x}$			$\boldsymbol{y}$							
x_2	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

$$y_0 = \overline{x_2}\ \overline{x_1}\ \overline{x_0} \quad y_4 = x_2\ \overline{x_1}\ \overline{x_0}$$
$$y_1 = \overline{x_2}\ \overline{x_1}\ x_0 \quad y_5 = x_2\ \overline{x_1}\ x_0$$
$$y_2 = \overline{x_2}\ x_1\ \overline{x_0} \quad y_6 = x_2\ x_1\ \overline{x_0}$$
$$y_3 = \overline{x_2}\ x_1\ x_0 \quad y_7 = x_2\ x_1\ x_0$$

(a) Tavola di verità di un decodificatore a 3 bit



(b) Circuito del decodificatore a 3 bit

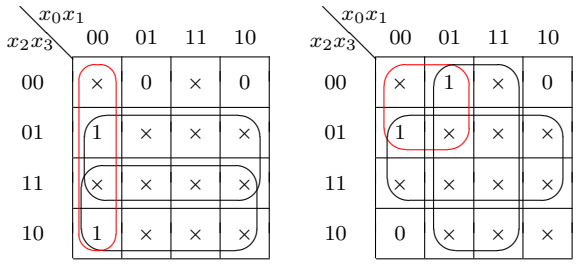
Figura 5.15: Tavola di verità e circuito di un decodificatore a 3 bit

5.4.2 Codificatori

I codificatori convertono un numero intero in base 10 in un numero espresso in notazione posizionale in base 2. Un codificatore svolge dunque la funzione inversa di un decodificatore, nel senso che esso prevede $m = 2^n$ ingressi e n uscite. Le uniche configurazioni ammesse per gli ingressi sono quelle in cui c'è esattamente un solo 1 in corrispondenza della linea x_j; dunque $x_j = 1$ e $x_i = 0$ per ogni $i \neq j$. Indicando con $x_0, x_1, ..., x_{m-1}$ gli ingressi, la corrispondente configurazione di uscita sulle variabili $y_{n-1}, y_{n-2}, ..., y_0$ è tale che esse codificano j in notazione posizionale in base 2. Nelle figure 5.16a e 5.16b vengono riportate, rispettivamente, la tavola di verità e le due mappe di Karnaugh associate alle variabili y_1 e y_0. Si noti che nella tabella di verità sono state riportate le sole configurazioni di ingresso definite; le altre danno luogo a condizioni non specificate, che consentono un amplissimo margine di libertà nella realizzazione effettiva del circuito. Ciò è evidente dalle mappe di Karnaugh di figura 5.16b, che consentono almeno 2 soluzioni; la prima implica l'uso di due porte OR e la seconda due porte AND e due NOT. Nella figura 5.17a e 5.17b vengono riprodotti, rispettivamente, il circuito del codificatore a 2 bit

$\boldsymbol{x}$				$\boldsymbol{y}$	
x_0	x_1	x_2	x_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

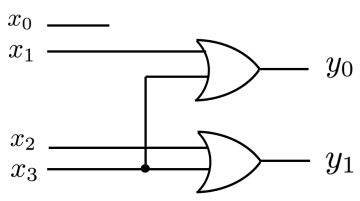
(a) Tavola di verità di un codificatore a 2 bit



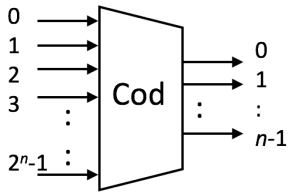
(b) Mappe di Karnaugh per le funzioni di figura 5.16a

Figura 5.16: Tavola di verità e mappe di Karnaugh del codificatore a 2 bit

associato alla soluzione con le porte OR e il simbolo circuitale di un generico codificatore a n bit. Analogamente, nelle figure 5.18a e 5.18b abbiamo la tavola di verità e il circuito di un codificatore a 3 bit.



(a) Circuito del codificatore a 2 bit



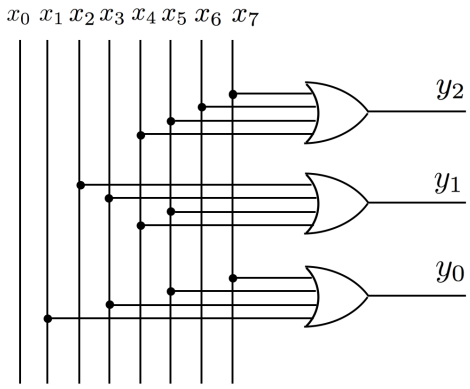
(b) Simbolo circuitale di un codificatore a n bit

Figura 5.17: Codificatore a 2 bit e simbolo circuitale di un decodificatore a n bit

$\boldsymbol{x}$								$\boldsymbol{y}$		
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	y_2	y_1	y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$y_2 = x_4 + x_5 + x_6 + x_7$$
$$y_1 = x_2 + x_3 + x_6 + x_7$$
$$y_0 = x_1 + x_3 + x_5 + x_7$$

(a) Tavola di verità di un codificatore a 3 bit



(b) Circuito del codificatore a 3 bit

Figura 5.18: Codificatore a 3 bit

5.4.3 Selettori

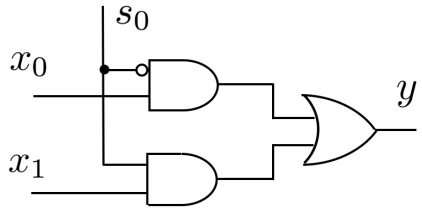
Un *selettore d'ingresso* (o *Multiplexer* o *Mux*) è un modulo che permette di selezionare uno tra 2^n ingressi e presentarlo sull'unica uscita. La selezione si effettua attraverso n linee di comando.

$\boldsymbol{s}$	$\boldsymbol{x}$		$\boldsymbol{y}$
s_0	x_0	x_1	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) Tavola di verità di un *Multiplexer* a 2 vie

$$\begin{aligned} y &= \overline{s_0}x_0\overline{x_1} + \overline{s_0}x_0x_1 \\ &+ s_0\overline{x_0}x_1 + s_0x_0x_1 \\ &= \overline{s_0}x_0 + s_0x_1 \end{aligned}$$

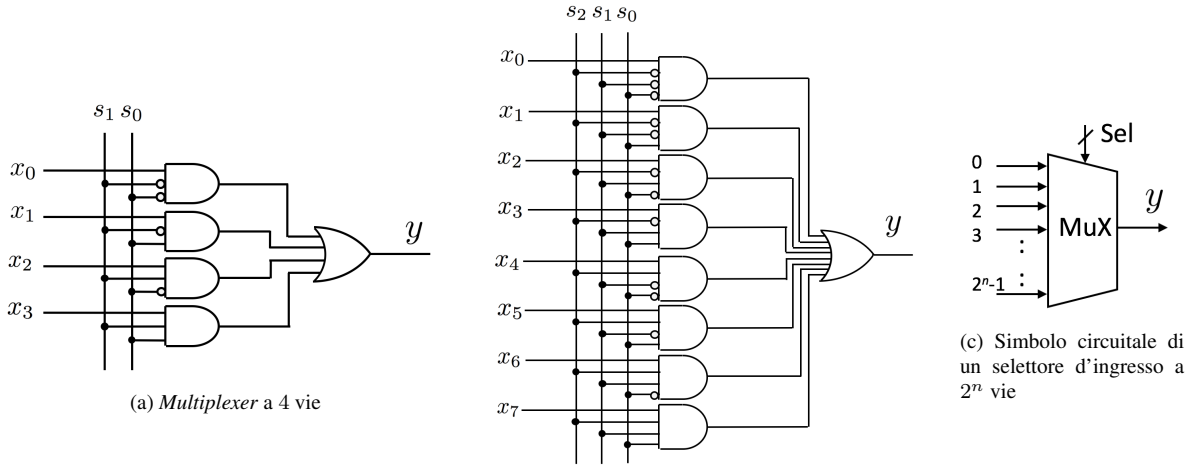
(b) *minterm* per la tavola di verità 5.19a



(c) *Multiplexer* a 2 vie basato sui *minterm* di figura 5.19b

Figura 5.19: Selettore d'ingresso (o *Multiplexer*) a 2

Nella figura 5.19a viene fornita la tavola di verità del *Multiplexer* a 2 vie (2:1 *Mux*), che si ricava tenendo conto che per $s_0 = 0$ viene riportato in uscita il contenuto di x_0, mentre per $s_0 = 1$ si porta in uscita il contenuto di x_1; la semplificazione tramite *minterm* di figura 5.19b porta alla soluzione circuitale di figura 5.19c. Nelle figure 5.20a, 5.20b sono invece rappresentati i *Multiplexer* a 4 e 8 vie, mentre la figura 5.20c illustra il simbolo circuitale di un generico *Multiplexer* a n vie.
Un *selettore d'uscita* (o *Demultiplexer* o *Demux*) è al contrario un dispositivo che permette di dirottare l'unico ingresso su una delle possibili 2^n uscite.

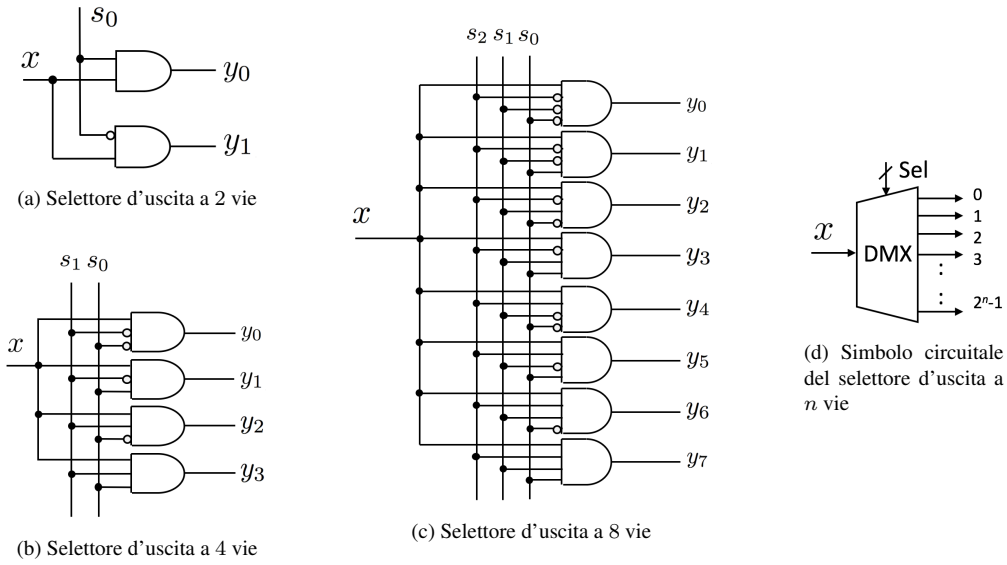


(a) *Multiplexer* a 4 vie

(b) Selettore d'ingresso a 8 vie

(c) Simbolo circuitale di un selettore d'ingresso a 2^n vie

Figura 5.20: Selettori d'ingresso a 4 e 8 vie e simbolo circuitale di un generico selettore a n vie



(a) Selettore d'uscita a 2 vie

(b) Selettore d'uscita a 4 vie

(c) Selettore d'uscita a 8 vie

(d) Simbolo circuitale del selettore d'uscita a n vie

Figura 5.21: Selettori d'uscita (o *demultiplexer*) a 2, 4 e 8 vie e relativo simbolo circuitale

La figura 5.21 mostra un *Demultiplexer* a 2 vie (1:2 *Demux*), uno a 4 vie (1:4 *Demux*) e uno a 8 vie (1:8 *Demux*), oltre al simbolo grafico di un generico selettore d'uscita a 2^n vie. Si noti che nel caso di selettore di uscita l'ingresso viene presentato sulla via selezionata, mentre tutti gli altri ingressi sono a 0.
In entrambi i selettori (d'ingresso e d'uscita) l'n-pla binaria della linea Sel seleziona quale, tra le 2^n linee, è interessata alla connessione con l'uscita (l'ingresso).

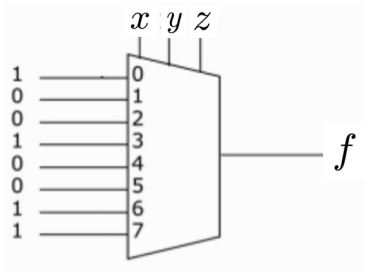
5.4.4 Costruzione modulare di una funzione Booleana

Illustriamo ora un impiego particolarmente interessante dei selettori d'ingresso, o *multiplexer*: la costruzione di qualunque funzione Booleana di n variabili attraverso un selettore d'ingresso a 2^n vie.
Una qualunque funzione di n variabili può essere ricondotta a una somma di *minterm*, come si è visto nell'equa-

zione (4.16). Usiamo come esempio la funzione Booleana di figura 5.22a.

		x	y	z	f	
m_0	$\overline{x}\overline{y}\overline{z}$	0	0	0	1	μ_0
m_1	$\overline{x}\overline{y}z$	0	0	1	0	μ_1
m_2	$\overline{x}y\overline{z}$	0	1	0	0	μ_2
m_3	$\overline{x}yz$	0	1	1	1	μ_3
m_4	$x\overline{y}\overline{z}$	1	0	0	0	μ_4
m_5	$x\overline{y}z$	1	0	1	0	μ_5
m_6	$xy\overline{z}$	1	1	0	1	μ_6
m_7	xyz	1	1	1	1	μ_7

(a) Funzione Booleana da realizzare col *Multiplexer*



(b) *Multiplexer* che realizza la funzione Booleana di figura 5.22a

Figura 5.22: Funzione Booleana e selettore idoneo a realizzarla

La funzione può essere scritta come

$$f(x,y,z) = \sum_{i\in\{0,3,6,7\}} m_i = m_0 + m_3 + m_6 + m_7 = \overline{x}\overline{y}\overline{z} + \overline{x}yz + xy\overline{z} + xyz$$
$$= 1\cdot\overline{x}\overline{y}\overline{z} + 0\cdot\overline{x}\overline{y}z + 0\cdot\overline{x}y\overline{z} + 1\cdot\overline{x}yz + 0\cdot x\overline{y}\overline{z} + 0\cdot x\overline{y}z + 1\cdot xy\overline{z} + 1\cdot xyz \qquad (5.4)$$

poiché $0, 3, 6$ e 7 sono le codifiche in base due di 000, 011, 110 e 111; nella 5.4 si mette 1 in corrispondenza dei *minterm* che compaiono nella funzione e 0 in corrispondenza di quelli che non compaiono. Con un selettore a otto vie la funzione f si ottiene impiegando x, y, z come linee di selezione e ponendo le otto linee di ingresso a 0 o a 1 a seconda che il coefficiente del corrispondente *minterm*, che corrisponde al valore della funzione per la terna binaria associata, sia a 0 o a 1. In figura 5.23 si vede il circuito per l'attuazione della funzione in oggetto. Poiché ogni

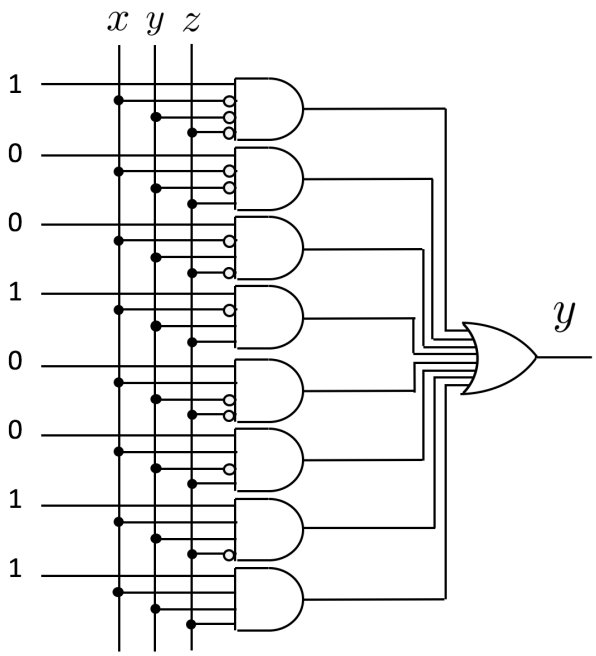
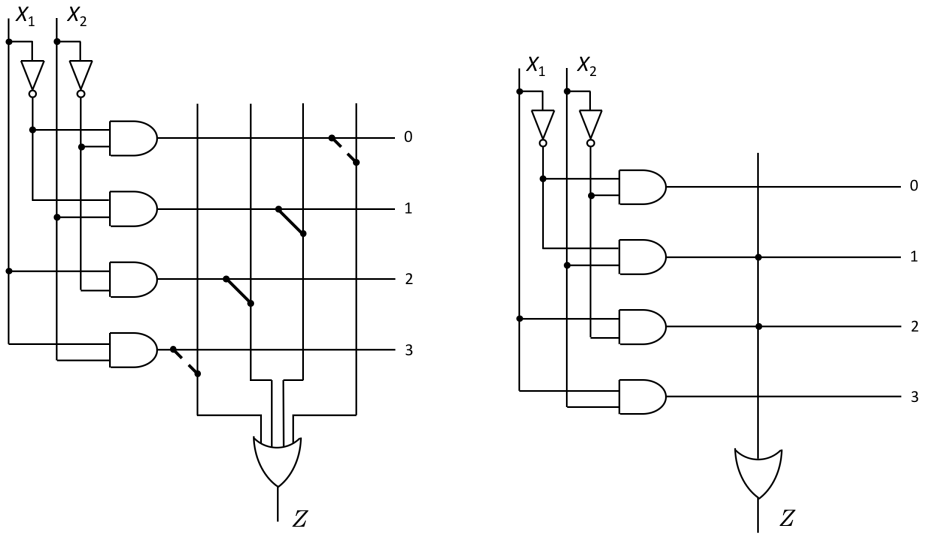


Figura 5.23: Modulo selettore per la realizzazione della funzione 5.4

funzione logica può essere ricondotta alla sua prima forma canonica, ne deriva che è possibile realizzare qualunque

funzione con il metodo sopra esposto.
E' ragionevole domandarsi perché usare tale metodo. La risposta sta nel fatto che con i circuiti integrati la soluzione con *multiplexer*, pur comportando solitamente un numero maggiore di porte, può portare a una riduzione del costo complessivo grazie alla produzione su larga scala di tali moduli. Inoltre, l'impiego del *multiplexer* dà maggiore flessibilità al progetto; infatti gli ingressi corrispondenti ai coefficienti possono essere considerati come ingressi di programmazione e possono essere aggiustati sulla piastra elettronica anche attraverso collegamenti ad hoc (verso $+V_{CC}$ o verso massa, che in logica positiva corrispondono rispettivamente a 1 e 0 logico). C'è tuttavia un metodo

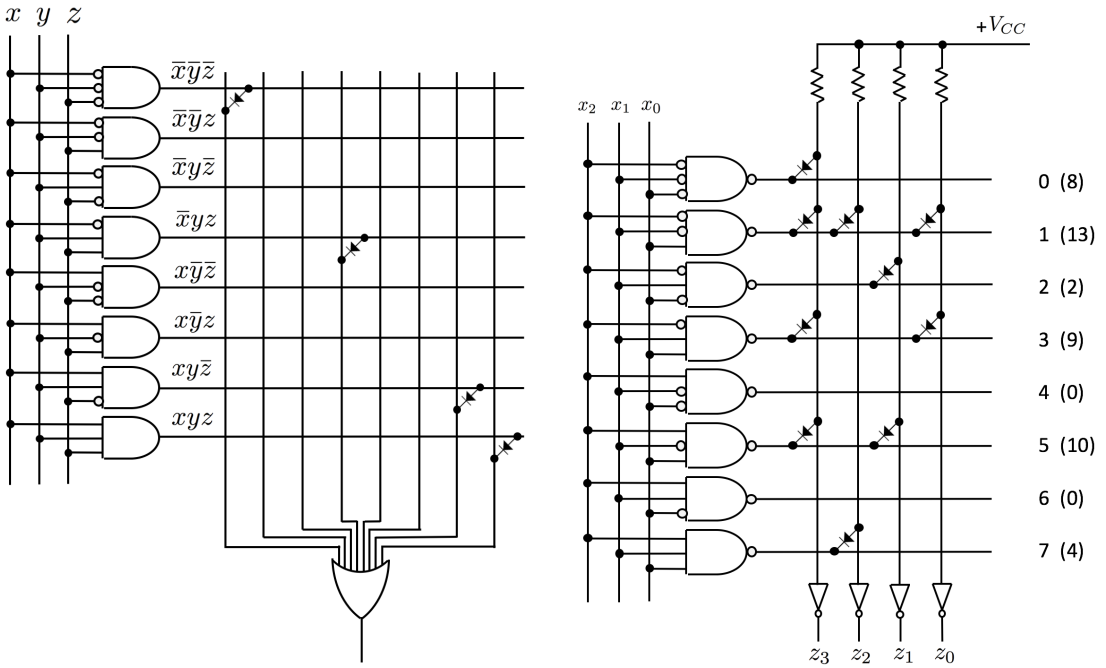


(a) Matrice di contatti a 2 bit

(b) Circuito effettivo per la matrice di contatti a 2 bit

Figura 5.24: Matrice di contatti a 2 bit

più economico per costruire, in modo modulare, una qualunque funzione Boolena di n variabili. Facendo sempre riferimento alla I forma canonica, la sua realizzazione richiede un numero di porte AND a n ingressi pari al numero di *minterm* (cioè 2^n) e una porta OR con un numero di ingressi pari al numero di porte AND (cioè sempre 2^n). Si tratta allora di "prolungare" le uscite delle porte AND agli ingressi della porta OR per i soli *minterm* che entrano nella funzione. La figura 5.24a mostra un circuito di questo genere, chiamato *matrice di contatti*, nel quale si evidenzia la presenza di un contatto per i soli *minterm* che servono per realizzare la funzione $z = \overline{x_1}\, x_2 + x_1\, \overline{x_2}$; nella pratica circuitale la configurazione diventa quella di figura 5.24b. Se ora vogliamo usare questa tecnica per costruire la funzione (5.4) otteniamo il circuito di figura 5.25a. La sua struttura è estremamente vantaggiosa se viene realizzata direttamente dal costruttore di integrati, che privilegia un'alta uniformità nella struttura circuitale. Il confronto con la soluzione a selettori di figura 5.23 è chiarificatore: in quel caso, per realizzare una qualunque funzione di tre variabili occorre un integrato che abbia almeno $3 + 8 + 1 = 12$ piedini (a parte l'alimentazione e la massa). Con la soluzione ora illustrata il numero di piedini sarebbe pari a soli $3 + 1 = 4$. La struttura di figura 5.25a fa uso di diodi per realizzare le connessioni della matrice; inoltre essa può essere estesa in modo tale da fornire non una, ma più funzioni di uscita, passando p.es. a una rete con n ingressi e k uscite. Una simile rete fornisce, per ogni configurazione degli n ingressi, una configurazione sulle k uscite e viene a costituire quella che si chiama una memoria *ROM*, acronimo di *Read Only Memory* (memoria di sola lettura); la memoria in questione ha $M = 2^n$ celle da k bit ciascuna, che vengono indirizzate dalle n linee di indirizzamento. Tenendo conto della realizzazione circuitale effettiva illustrata in figura 5.24b, riportiamo in figura 5.25b la struttura di una memoria ROM con 2^3 celle di memoria da 4 bit ciascuna.
Nella parte destra della figura 5.25b viene riportato l'indirizzo di ciascuna cella e, tra parentesi, il suo contenuto. Per capirne il funzionamento supponiamo che si voglia leggere il contenuto della memoria all'indirizzo 3; ciò significa che deve essere $x_2x_1x_0 = 011$. In tal caso ci sarà una sola porta NAND che ha un'uscita bassa, la porta della riga 3, mentre tutte le altre porte NAND avranno uscita alta. I catodi dei diodi relativi alle due colonne z_3 e



(a) Matrice di contatti a 4 bit che attua la funzione (5.4)

(b) Memoria ROM con 2^3 celle di memoria da 4 bit ciascuna

Figura 5.25: Matrice di contatti e memoria ROM derivabile da essa

z_0 vanno allora a massa, inducendo una polarizzazione diretta per i corrispondenti diodi; sui rispettivi anodi si ha allora una tensione virtualmente nulla (in pratica ci sarà solo la V_{AK} di saturazione, pari $0,3 \div 0,6\ V$). Le linee z_3 e z_0 sono allora basse, e a seguito della complementazione finale diventano alte. Gli altri diodi sulle colonne z_3 e z_0 risultano invece interdetti, le linee rimangono allo stato alto ($+V_{CC}$) e quindi c'è 0 in uscita, a seguito della complementazione finale. Dunque $z_3 = 1$ e $z_0 = 1$, mentre le altre due linee z_2 e z_1 si trovano a zero, e dunque $\boldsymbol{z} = 1001$.

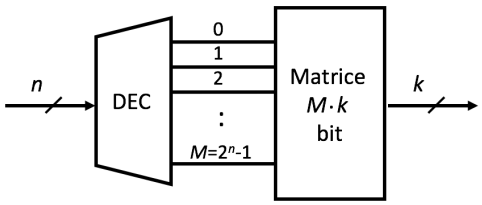


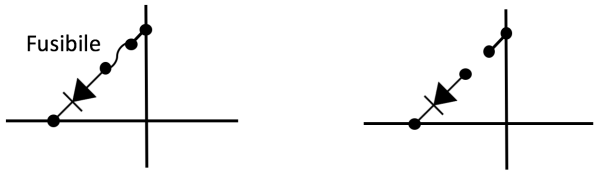
Figura 5.26: Schema a blocchi di una memoria ROM

In figura 5.26 viene dato lo schema a blocchi di una memoria ROM; di fatto si tratta di un decodificatore seguito da una matrice di contatti. Il numero binario corrispondente alla combinazione degli n ingressi rappresenta l'indirizzo della corrispondente cella.

Si noti che fissare i contatti della matrice equivale a programmare il comportamento della rete. Nelle ROM propriamente dette, la matrice non ha inizialmente alcun punto di contatto tra righe e colonne. E' il costruttore che fa le connessioni in base alla matrice di bit desiderata dal committente. Per le memorie ROM, che sono appunto di sola lettura, non è possibile variare il contenuto della memoria dopo la programmazione fatta in fabbrica. Le ROM risultano economicamente convenienti per volumi molto grandi.

Nel caso debba essere l'utente a programmare la ROM si ricorre alle cosiddette *PROM*(*Programmable Read Only Memory*); la matrice ha inizialmente tutti i punti di contatto tra righe e colonne, attraverso un diodo e un fusibile,

come schematizzato a destra in figura 5.27a, e dunque inizialmente tutti i bit sono a 1. Programmare un bit a 0 ri-



(a) Connessione con diodo prima della programmazione

(b) Connessione con diodo dopo la programmazione del bit a 0

Figura 5.27: Modalità di interconnessione tra righe e colonne per le PROM prima e dopo la programmazione di un bit a 0

chiede la fusione del relativo fusibile. Spetta all'utente inserire gli 0 negli incroci desiderati, attraverso un apparato di programmazione. Chiaramente non è possibile modificare il contenuto della memoria dopo la programmazione. Le PROM risultano economicamente convenienti per volumi medio/grandi.
Esiste anche la possibilità di modificare la programmazione ricorrendo alle *EPROM* (*Erasable* PROM), basate sulla tecnologia dei *Floating Gate MOSFET*; la memoria è in un contenitore che presenta una piastrina di quarzo, attraverso la quale possono passare raggi ultravioletti che ripristinano la programmabilità cancellando la programmazione precedente (si veda la figura *fig:EPROM*). Le EPROM risultano convenienti per prototipi di laboratorio o per bassissimi volumi di produzione. Sulla base della tecnologia EPROM, nel 1978 vennero sviluppate delle me-

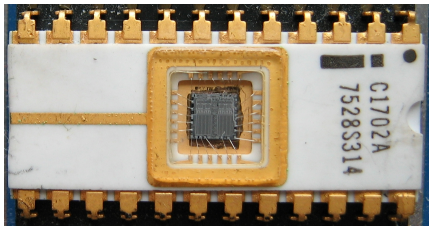


Figura 5.28: La prima EPROM realizzata da INTEL nel 1971, da 256 byte

morie cancellabili e riprogrammabili elettricamente, le cosiddette *EEPROM* (*Electrically Erasable* PROM), che fanno uso di MOSFET con uno strato molto più sottile di ossido per il gate. Queste memorie sono indicate per un impiego che prevede la possibilità di modificarne il contenuto direttamente dall'apparato in cui vengono usate.

5.5 Moduli per la realizzazione dell'unità logico-aritmetica

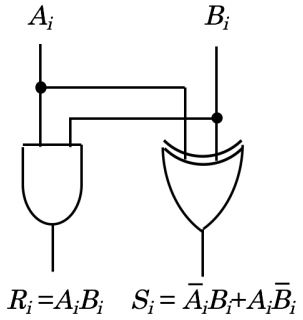
Come vedremo nel capitolo 7, uno degli elementi centrali nell'architettura di un calcolatore è l'*Unità Logico Aritmetica*, meglio nota con l'acronimo *ALU* - che sta per *Arithmetic Logic Unit*; essa lavora a stretto contatto con i registri di memoria, guidata dall'azione dell'*Unità di Controllo*. Nella ALU si realizzano tipicamente operazione su numeri interi quali somma, sottrazione, incremento, decremento, scorrimento di bit, ma si attuano anche operazioni logiche sui dati, quali AND, OR, XOR o complementazione; le ALU più recenti contengono anche moduli per eseguire direttamente prodotti e moltiplicazioni.

5.5.1 Il semisommatore e il sommatore completo

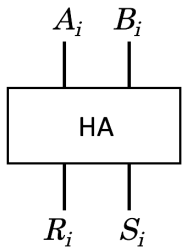
Il nucleo di partenza per costruire una ALU è il modulo *semisommatore* (o *Half Adder*), che realizza la somma di due bit con riporto. In figura 5.29a è riprodotta la tabella aritmetica della somma bit per bit con riporto,

A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) Tavola di verità del semisommatore



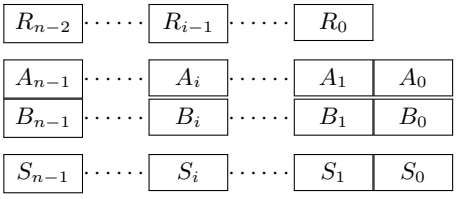
(b) Circuito del semisommatore



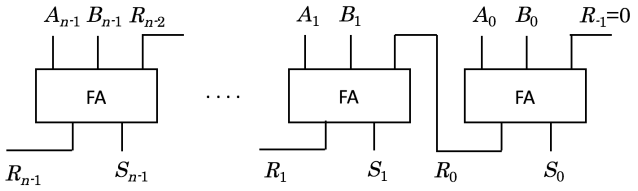
(c) Simbolo circuitale del semisommatore

Figura 5.29: Il semisommatore

che corrisponde alla tavola di verità di due funzioni Booleane, una che realizza la somma S_i e una che realizza il riporto R_i, per ogni coppia A_i, B_i dei bit d'ingresso. Si riconosce immediatamente che la S_i corrisponde a uno XOR, mentre la R_i è un AND; di conseguenza il modulo ha la realizzazione circuitale di figura 5.29b, mentre la figura 5.29c rappresenta il suo simbolo circuitale.
Se ora vogliamo effettuare la somma completa tra due numeri interi $\boldsymbol{A} = [A_{n-1}\ A_{n-2}\ ...\ A_1\ A_0]$ e $\boldsymbol{B} = [B_{n-1}\ B_{n-2}\ ...\ B_1\ B_0]$, espressi in notazione posizionale con n bit, dobbiamo costruire una rete che accetti in ingresso $2n$ bit e generi la somma binaria dei due, $\boldsymbol{S} = [S_{n-1}\ S_{n-2}\ ...\ S_1\ S_0]$, secondo il classico procedimento di somma con riporto evidenziato in figura 5.30a. Partendo da destra si inizia a sommare A_0 con B_0; detto R_0 il



(a) Procedimento di somma con riporto tra due numeri interi $\boldsymbol{A} = [A_{n-1}\ A_{n-2}\ ...\ A_1\ A_0]$ e $\boldsymbol{B} = [B_{n-1}\ B_{n-2}\ ...\ B_1\ B_0]$



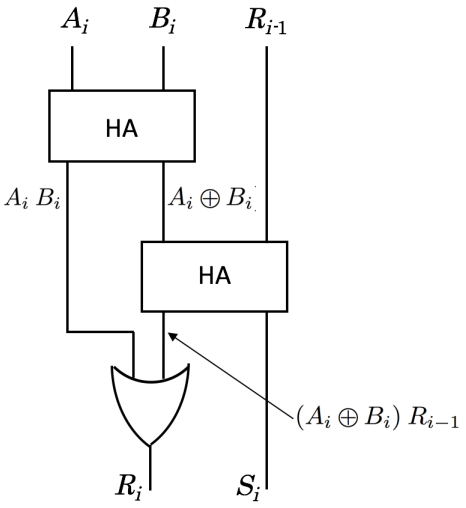
(b) Sommatore completo

Figura 5.30: Somma con riporto e relativo circuito sommatore

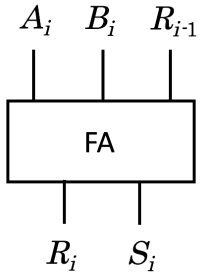
riporto, questi dovrà essere sommato con la somma di A_1 e B_1, che genera R_1; questo va sommato con la somma di A_2 e B_2 e così via. E' evidente che il riporto R_{-1} della colonna $A_0 \| B_0$ vale 0 e che se l'ultimo riporto, R_{n-1} dovesse valere 1, siamo di fronte a una situazione di *overflow*, che richiede il passaggio alla notazione a virgola mobile. Per realizzare un tale circuito bisogna modificare il semisommatore, in modo da includere il contributo R_{i-1} del riporto del passo precedente. In questo modo si ottiene il *sommatore completo* (o *full adder*), la cui tavola di verità è riportata in figura 5.31a. La somma di A_i, B_i e R_{i-1} vale 1 solo quando c'è un numero dispari di 1 nella somma, ed è quindi lo XOR dei tre bit; R_i vale 1 quando A_i e B_i sono entrambi a 1 (qualunque sia il valore di R_{i-1}), oppure quando $R_{i-1} = 1$ e $A_i \oplus B_i = 1$. Per ricavare in modo formale l'espressione risolutiva scriviamo i *minterm* di entrambe le funzioni che si ricavano dalla tavola di verità di figura 5.31a e procediamo con

A_i	B_i	R_{i-1}	S_i	R_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Tavola di verità del sommatore completo



(b) Circuito del sommatore completo



(c) Simbolo circuitale del sommatore completo

Figura 5.31: Il sommatore completo

le semplificazioni, ricordando che $x \oplus y = x\overline{y} + \overline{x}y$ e che $\overline{x \oplus y} = \overline{x}\overline{y} + xy$

$$\begin{aligned}
S_i &= \overline{A_i}\,\overline{B_i}R_{i-1} + \overline{A_i}B_i\overline{R_{i-1}} + A_i\overline{B_i}\,\overline{R_{i-1}} + A_iB_iR_{i-1} \\
&= (\overline{A_i}\,\overline{B_i} + A_iB_i)R_{i-1} + (\overline{A_i}B_i + A_i\overline{B_i})\overline{R_{i-1}} \\
&= (\overline{A_i \oplus B_i})R_{i-1} + (A_i \oplus B_i)\overline{R_{i-1}} \\
&= (A_i \oplus B_i) \oplus R_{i-1} \\
R_i &= \overline{A_i}B_iR_{i-1} + A_i\overline{B_i}R_{i-1} + A_iB_i\overline{R_{i-1}} + A_iB_iR_{i-1} \\
&= (\overline{A_i}B_i + A_i\overline{B_i})R_{i-1} + A_iB_i \\
&= (A_i \oplus B_i)\ R_{i-1} + A_i\ B_i
\end{aligned} \tag{5.5}$$

e da queste equazioni si ricava direttamente la struttura circuitale di figura 5.31b, che viene rappresentata simbolicamente come in figura 5.31c. Tale realizzazione ha anche il pregio di sfruttare la modularità del semisommatore.

5.5.2 Calcolo della differenza mediante sommatore

Nel paragrafo 2.3.2 si è visto che i numeri negativi vengono rappresentati con la notazione mediante complemento a 2. Ciò significa che la differenza $\boldsymbol{A}-\boldsymbol{B}$ si realizza come $\boldsymbol{A}+(-\boldsymbol{B})$, dove $-\boldsymbol{B}$ si ottiene complementando $\boldsymbol{B}$ e sommando 1. Si ha dunque

$$\boldsymbol{A} - \boldsymbol{B} = \boldsymbol{A} + (-\boldsymbol{B}) = \boldsymbol{A} + \overline{\boldsymbol{B}} + 1 \tag{5.6}$$

Il circuito di figura 5.32 offre la possibilità di fare la differenza tra $\boldsymbol{A}$ e $\boldsymbol{B}$ secondo il principio appena esposto; il funzionamento è il seguente. Il valore assunto dalla linea di controllo C_B determina se il valore $\boldsymbol{B}$ viene o meno complementato prima di entrare nel sommatore; infatti se $C_B = 0$ si attivano gli AND sulla sinistra, che fanno passare $\boldsymbol{B}$; se invece $C_B = 1$ si attivano gli AND sulla destra, che fanno passare il suo complementare $\overline{\boldsymbol{B}}$. Se vogliamo fare la differenza $\boldsymbol{A} - \boldsymbol{B} = \boldsymbol{A} + (-\boldsymbol{B}) = \boldsymbol{A} + \overline{\boldsymbol{B}} + 1$, dobbiamo allora porre $C_B = 1$ e aggiungere 1 tramite R_{-1}. La linea di controllo C_A serve invece per azzerare l'ingresso $\boldsymbol{A}$; infatti se $C_A = 1$, in ingresso al sommatore viene mandato $\boldsymbol{A}$, altrimenti si hanno degli 0. La tabella 5.33 mostra il risultato delle varie combinazioni di C_B, C_A, R_{-1}. Le tre porte di sinistra di figura 5.32 servono invece per dare un segnale di allarme quando si è in presenza di *overflow*; infatti dalla tavola 2.14 si può osservare che tale condizione si realizza quando si verificano le seguenti condizioni:

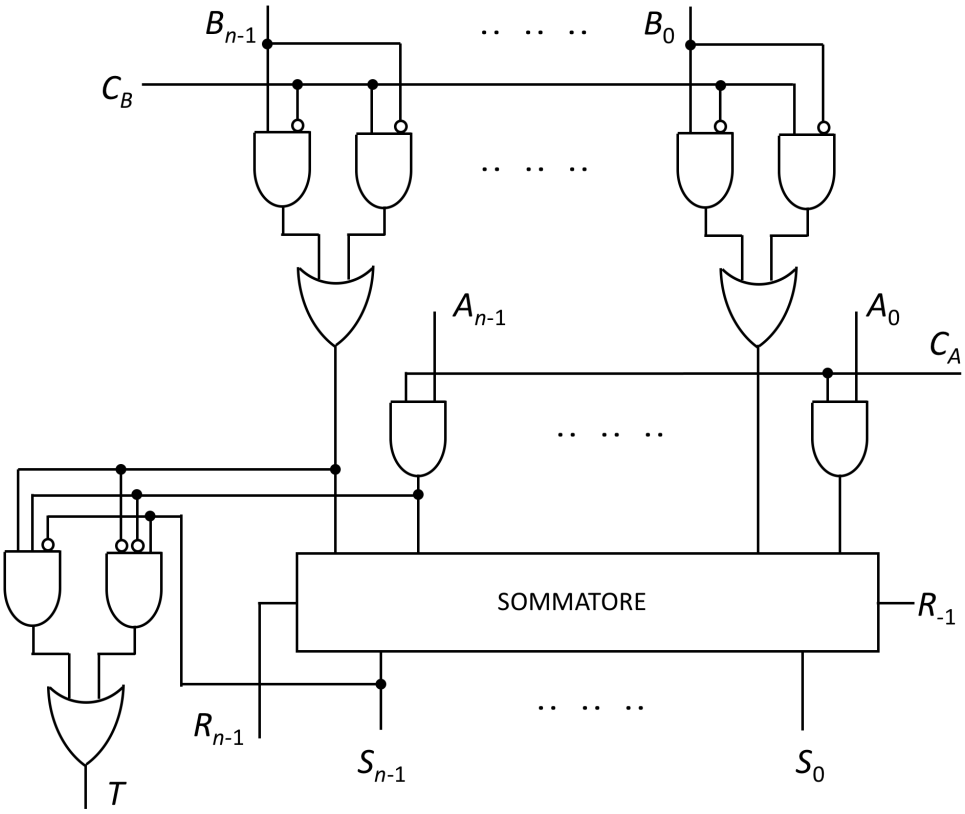


Figura 5.32: Circuito sommatore che può eseguire anche la differenza

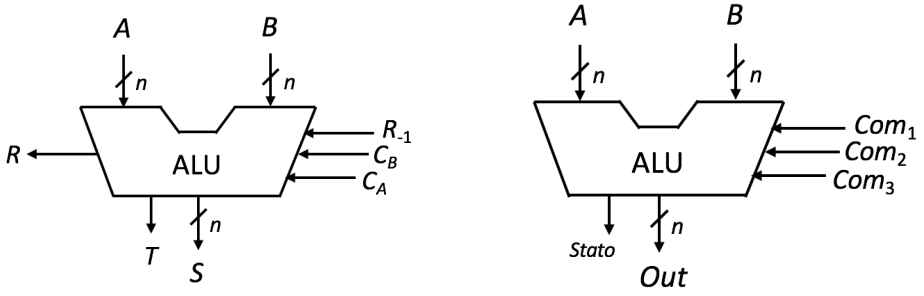
- si sommano due numeri negativi (cioè con 1 nella prima posizione) e il risultato è positivo (0 nella prima posizione);

- si sommano due numeri positivi (0 nella prima posizione) e il risultato è negativo (cioè con 1 nella prima posizione)

La prima delle due condizioni si ha con $A_{n-1} = 1$, $B_{n-1} = 1$, $S_{n-1} = 0$, e quando si realizza fornisce un uscita 1 nella porta AND di sinistra; la seconda condizione si ha con $A_{n-1} = 0$, $B_{n-1} = 0$, $S_{n-1} = 1$ e quando si realizza fornisce un uscita 1 nella porta AND di destra. La porta OR raccoglie dunque l'unione logica dei due eventi. Il segnale di allarme viene usato per commutare nella rappresentazione a virgola mobile.
In figura 5.34a viene illustrato il simbolo schematico della ALU descritta dalla rete 5.32. Con una piccola modifica della circuiteria è possibile fare in modo da realizzare, oltre alla somma e alla differenza tra $\boldsymbol{A}$ e $\boldsymbol{B}$, anche l'AND e l'OR tra i due ed eventuali altre operazioni logiche; lo schema di figura 5.34b mostra una ALU completa, nella quale i comandi C_A, C_B, R_{-1} e tutti gli altri per attivare le varie operazioni logiche sono rappresentati con la notazione $\text{Com}_1, \text{Com}_2, ..., \text{Com}_n$. Invitiamo il lettore interessato ad approfondire la questione sul testo di Bucci [2].

C_A	C_B	R_{-1}	Operazione	Descrizione
0	0	0	$\boldsymbol{S} = 0 + \boldsymbol{B}$	Selezione di $\boldsymbol{B}$
0	0	1	$\boldsymbol{S} = 0 + \boldsymbol{B} + 1 = \boldsymbol{B} + 1$	Incremento di $\boldsymbol{B}$
0	1	0	$\boldsymbol{S} = 0 + \overline{\boldsymbol{B}} = \overline{\boldsymbol{B}}$	Complementazione di $\boldsymbol{B}$
0	1	1	$\boldsymbol{S} = 0 + \overline{\boldsymbol{B}} + 1 = -\boldsymbol{B}$	Cambio segno di $\boldsymbol{B}$
1	0	0	$\boldsymbol{S} = \boldsymbol{A} + \boldsymbol{B}$	Somma $\boldsymbol{A} + \boldsymbol{B}$
1	0	1	$\boldsymbol{S} = \boldsymbol{A} + \boldsymbol{B} + 1$	
1	1	0	$\boldsymbol{S} = \boldsymbol{A} + \overline{\boldsymbol{B}} = \boldsymbol{A} - \boldsymbol{B} - 1$	
1	1	1	$\boldsymbol{S} = \boldsymbol{A} + \overline{\boldsymbol{B}} + 1 = \boldsymbol{A} - \boldsymbol{B}$	Differenza $\boldsymbol{A} - \boldsymbol{B}$

Figura 5.33: Operazioni effettuate dalla rete di figura 5.32 a seconda dei valori assunti dagli ingressi di controllo C_A, C_B, R_{-1}



(a) Simbolo schematico della ALU descritta dalla rete 5.32

(b) Simbolo schematico di una generica ALU dotata di linee di comando per generare operazioni logiche tra gli ingressi

Figura 5.34: Simbolo schematico di una ALU

Le moderne ALU vengono integrate con circuiti che realizzano la moltiplicazione e la divisione tra numeri interi, anche per questi circuiti dedicati invitiamo a consultare [2].

Capitolo 6

Circuiti sequenziali

6.1 Introduzione

I circuiti considerati fino a questo punto sono i circuiti combinatori, nei quali in ogni istante la configurazione di una generica variabile di uscita y_i dipende unicamente dal valore assunto dalle variabili d'ingresso $x_1, x_2, ..., x_n$, secondo la funzione Booleana $y_i = f_i(x_1, x_2, ..., x_n)$, $1 \leqslant i \leqslant m$. Il modello generale di un circuito combinatorio è illustrato in figura 6.1; si noti che in esso non compare la variabile temporale, a sottolineare il

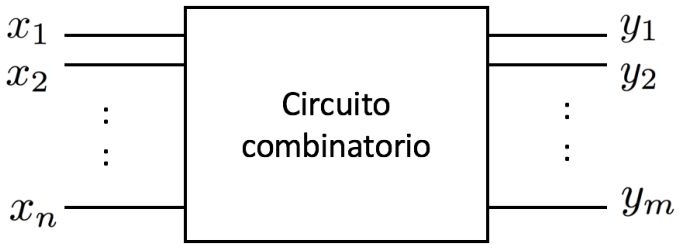


Figura 6.1: Schema generale di un circuito di commutazione con n ingressi e m uscite

fatto che i valori presi in considerazione per le n variabili di ingresso si riferiscono allo stesso istante e il comportamento della rete, se escludiamo i fenomeni transitori, è univocamente dedotto dalla tavola di verità che stabilisce il legame tra ciascuna y_i e le variabili d'ingresso $x_1, x_2, ..., x_n$.
Nel caso in cui in un sistema il valore delle uscite dipenda anche dalla storia passata della circuiteria che lo costituisce, ovvero dallo *stato* della rete, si parla di sistema o circuito *sequenziale*. Il sistema telefonico è un tipico esempio di sistema sequenziale: se infatti si è in procinto di comporre l'ultima cifra di un numero telefonico, il comportamento del sistema dipenderà anche dalle cifre precedentemente selezionate; questa cifra è l'ingresso attuale del sistema e l'uscita sarà il segnale che effettuerà il collegamento. Ovviamente l'ingresso attuale non è il solo fattore che determina il collegamento, poiché anche le cifre composte precedentemente sono ugualmente importanti. Anche un calcolatore elettronico è un esempio di circuito sequenziale, anzi è l'esempio per eccellenza; di solito in esso vengono usati in maniera sequenziale diversi sottoinsiemi che possono essere di volta in volta sequenziali o combinatori.
Da un punto di vista formale un circuito sequenziale è un *automa a stati finiti* $\mathcal{M}$, cioè un sistema dinamico *di-*

screto (nella scansione del tempo e nella descrizione del suo stato) e *stazionario* (il sistema si comporta alla stessa maniera indipendentemente dall'istante di tempo in cui agisce). Esso è caratterizzato da:

- un insieme finito $\mathcal{Q} = \{q_1, q_2, ..., q_S\}$ di stati interni;
- un insieme finito $\mathcal{A} = \{a_1, a_2, ..., a_K\}$ di valori che possono essere assunti dalle variabili d'ingresso $\boldsymbol{x} = \{x_1, x_2, ..., x_n\}$;
- un insieme finito $\mathcal{B} = \{b_1, b_2, ..., b_D\}$ di valori che possono essere assunti dalle variabili di uscita $\boldsymbol{y} = \{y_1, y_2, ..., y_m\}$;
- un insieme di regole, detto *mappa di transizione* τ, che specifica lo stato q^* raggiunto dalla macchina a partire dallo stato q per effetto dell'ingresso $\boldsymbol{x}$;
- un insieme di regole, detto *mappa delle uscite* $\mathcal{U}$, che specifica il valore $\boldsymbol{y}^*$ assunto dalle variabili di uscita $\{y_1, y_2, ..., y_m\}$ per effetto dell'ingresso $\boldsymbol{x}$ applicato allo stato q.

La macchina sequenziale è pertanto definita dai cinque insiemi citati:

$$\mathcal{M} = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, \tau, \mathcal{U})$$

Lo studio sistematico della teoria degli automi verrà fatto nell'ambito del corso di *Complessità e Crittografia* della laurea magistrale, ma si è ritenuto in ogni caso utile introdurre il formalismo per inquadrare meglio il problema. Tanto per fare un breve esempio si consideri la seguente tabella

	ingresso	
stato	0	1
q_1	$q_1/0$	$q_2/1$
q_2	$q_2/1$	$q_3/1$
q_3	$q_1/0$	$q_3/1$

che esprime il funzionamento di un automa con un ingresso e un'uscita binari, e con tre stati interni. La tabella si legge in questo modo: se l'automa si trova nello stato q_1 e si applica 0 in ingresso, allora l'automa rimane nello stato q_1 e manda 0 in uscita; se viceversa si applica 1 in ingresso, allora l'automa passa nello stato q_2 e manda 1 in uscita. La stessa tecnica di lettura viene usata anche per gli altri stati. Poiché per ciascuno degli stati q_1, q_2, q_3 dell'automa la tabella specifica il comportamento dello stesso a seconda che l'ingresso sia 0 o 1, essa incorpora la mappa di transizione τ e la mappa delle uscite $\mathcal{U}$; costituisce dunque una forma di rappresentazione di un automa. Il funzionamento di una rete sequenziale può essere schematizzato come in figura 6.2, cioè come composizione di una rete combinatoria e di una memoria. Chiudiamo questa introduzione ricordando che nel nostro caso gli alfabeti d'ingresso $\mathcal{I}$ e d'uscita $\mathcal{W}$ saranno sempre binari.

6.2 Moduli sequenziali asincroni

Nei sistemi sequenziali introdotti al paragrafo precedente, l'informazione sulla storia passata del circuito, ovvero il valore acquisito dalla variabile di stato $\mathcal{Q}$, deve essere memorizzato su un qualche supporto. Si possono avere diversi tipi di dispositivi di memorizzazione, ma uno dei più usati è il cosiddetto *flip-flop*. In figura 6.3 è illustrato il *flip-flop* già introdotto nella figura 2.26, nel quale l'uscita di un transistor in interdizione viene connessa con l'ingresso di un transistor in conduzione. Il circuito che si ottiene è bistabile, nel senso che esso può stare indifferentemente e stabilmente in uno o nell'altro dei due stati rappresentati in figura 6.3a e 6.3b. Il funzionamento è basato sul fatto che, quando il transistor di sinistra è interdetto (non passa corrente nel circuito di collettore),

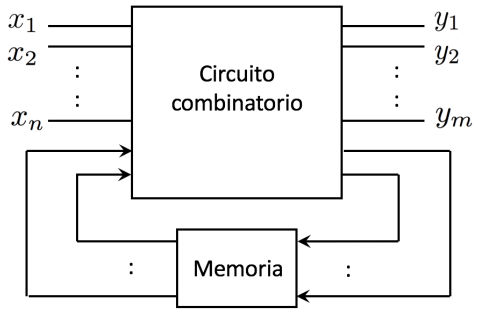
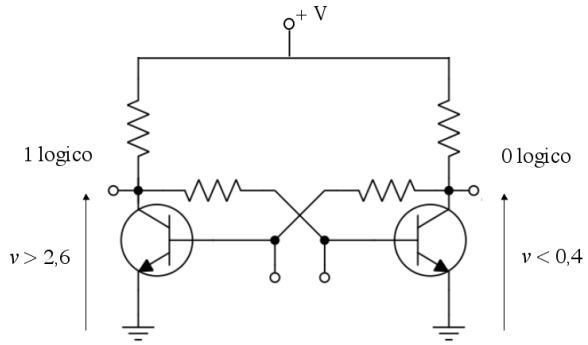
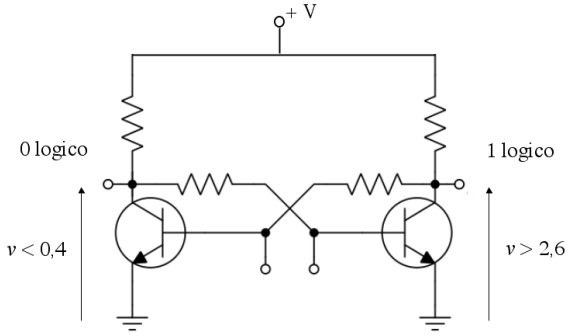


Figura 6.2: Schema logico di una rete sequenziale



(a) Il transistor di sinistra del *Flip-Flop* è interdetto; ciò porta in conduzione il transistor di destra, che fornisce uscita logica 0



(b) Il transistor di sinistra del *Flip-Flop* è in conduzione; ciò porta in interdizione il transistor di destra, che fornisce uscita logica 1

Figura 6.3: Circuito bistabile del *Flip-Flop*

allora la sua tensione di collettore è alta ($v > 2,6$ V); ciò polarizza la base del transistor di destra, che entra in piena conduzione, facendo collassare a un valore basso ($v < 0,4$ V) la sua tensione di uscita (figura 6.3a), che corrisponde a uno 0 logico. I ruoli dei due transistor si scambiano quanto la tensione di collettore del transistor di destra viene portata (con un impulso) a un valore ($v > 2,6$ V) (figura 6.3b). Dal funzionamento del circuito risulta evidente che esso costituisce un elemento di memoria di 1 bit; infatti si può decidere che l'uscita del collettore di uno dei due transistor rappresenti la variabile logica il cui valore vogliamo memorizzare; scegliendo p.es. il transistor di destra, se vogliamo memorizzare 1 dobbiamo mandare lo stesso transistor in interdizione; per lo 0 dobbiamo mandarlo in conduzione (in una logica positiva).

6.2.1 Il *Flip-Flop Set-Reset* - FFSR

Anche se perfettamente funzionante, questa realizzazione non viene usata nella pratica per memorizzare *bit*, poiché si preferisce sempre ricorrere alle porte logiche, che costituiscono le unità elementari di qualunque circuito logico. Il vantaggio di tale approccio consiste nel fatto che, stando all'interno di una certa famiglia logica, tutti i segnali di comando e i livelli di tensione sono uniformati per l'intero circuito; possiamo così aggiungere singole unità funzionali senza preoccuparci di uniformare i livelli di tensione, poiché sono già standardizzati all'interno della famiglia. Mostreremo allora le due realizzazioni principali del cosiddetto *Flip-Flop Set-Reset* (FFSR), basate rispettivamente sulle porte NOR e sulle porte NAND.

Latch di NOR

La figura 6.4 illustra un *flip-flop* realizzato con due porte NOR; in gergo viene anche chiamato *Latch* di NOR. La prima cosa che balza all'occhio è il fatto che entrambe le uscite X e Y sono riportate all'ingresso; questo

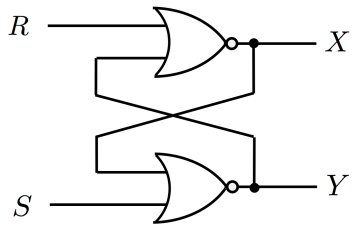


Figura 6.4: Flip-flop realizzato con due porte NOR

lascia presagire che il valore assunto dalle variabili di uscita dipenda anche dalle uscite stesse. Se impostiamo le equazioni del sistema otteniamo

$$
\begin{aligned}
&X = \overline{R+Y} \qquad Y = \overline{S+X} \qquad \text{e sostituendo} \\
&X = \overline{R+\overline{S+X}} \\
&\quad = \overline{R}\cdot\overline{\overline{S+X}} \qquad \text{De Morgan} \\
&\quad = \overline{R}\cdot(S+X) \qquad (6.1)
\end{aligned}
$$

che conferma la nostra previsione. Nonostante l'equazione (6.1) sia impeccabile, non ci rende conto chiaramente del comportamento di questa semplice rete. La cosa migliore da fare è allora quella di fissare i valori di S, R, X e Y in tutti i modi possibili e vedere quali quaterne sono compatibili con i vincoli imposti dalle equazioni 6.1. Nella figura 6.5a vengono riportate tutte le possibili combinazioni per R, S, X e Y; quelle in rosso non soddisfano l'equazione 6.1, perché $X \neq \overline{R}(S+X)$ oppure $Y \neq \overline{S+X}$; queste configurazioni non sono stabili. Nella successiva figura 6.5b si riportano invece i soli stati stabili, per semplicità di lettura. Si osservi che, a parte il caso in cui $R = S = 1$, che escludiamo per i motivi che vedremo nel seguito, in tutte le altre combinazioni lecite si ha sempre $X = \overline{Y}$, cioè X e Y sono l'uno complementare dell'altro.
Partiamo ora dalla condizione $R = S = 0$; dalla tabella 6.5b osserviamo che ci sono due stati stabili possibili, uno con $X = 0, Y = 1$ e l'altro con $X = 1, Y = 0$; supponiamo di essere nel secondo, cioè $X = 1, Y = 0$, così come evidenziato in figura 6.6a. Supponiamo ora di portare l'ingresso R da 0 a 1 nell'istante t_1; quando ciò avviene, l'uscita della porta 1 commuta a $X = 0$ con un certo ritardo τ, legato ai tempi di commutazione dei transistor della porta. Il nuovo segnale $X = 0$ alimenta l'ingresso della porta 2, facendo commutare Y a 1 con un ritardo pari a 2τ. Se ora riportiamo R a 0 (si veda figura 6.6b), X e Y rimangono nella stessa configurazione acquisita, cioè $X = 0$ e $Y = 1$, poiché essa è stabile rispetto a $R = S = 0$, a norma della tabella 6.5b. Riportando ora R nuovamente a 1, non cambia comunque nulla, perchè con $X = 0$ e $Y = 1$, R e S possono stare stabilmente in ciascuno dei due stati $R = S = 0$ oppure $R = 1, S = 0$. Quello che è successo è che inviando un impulso sull'ingresso R, che viene chiamato impulso di *Reset*, l'ingresso X va (o permane) a 0.

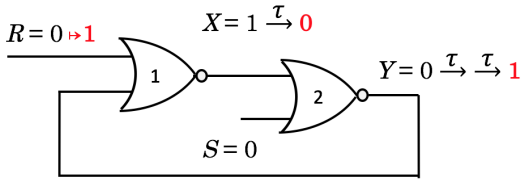
R	S	X	Y	$\overline{R}(S+X)$	$\overline{S+X}$
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

(a) Tavola di verità dell'equazione 6.1 per tutti gli stati possibili

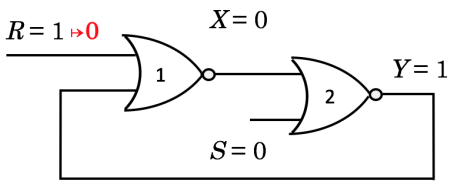
R	S	X	Y	$\overline{R}(S+X)$	$\overline{S+X}$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	0	0	0

(b) Tavola di verità dell'equazione 6.1 riferita ai soli stati stabili, cioè quelli per i quali $X = \overline{R}(S+X)$ e $Y = \overline{S+X}$

Figura 6.5: Tavola di verità dell'equazione 6.1

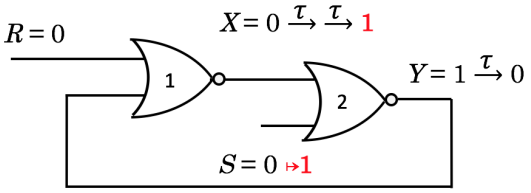


(a) *Reset* del *flip-flop*, che si attua ponendo $R = 1$

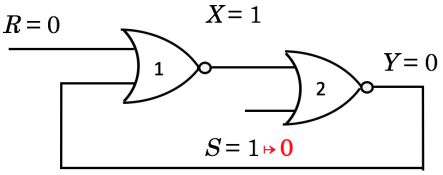


(b) Una volta attuato il *Reset*, anche riportando a R a 0 lo stato non cambia

Figura 6.6: *Reset* del *flip-flop*



(a) *Set* del *flip-flop*, che si attua ponendo $S = 1$



(b) Una volta attuato il *Set*, anche riportando a S a 0 lo stato non cambia

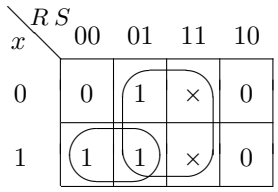
Figura 6.7: *Set* del *flip-flop*

Rifacciamo ora il ragionamento partendo dall'altra condizione, quella in cui con $R = S = 0$ si ha $X = 0, Y = 1$ (figura 6.7a,). Se portiamo S da 0 a 1 nell'istante t_1 (sempre con $X = 0$), l'uscita della porta 2 passa a 0 con un ritardo τ; istantaneamente tale $Y = 0$ si propaga all'ingresso della porta 1, e determina la commutazione di X a 1 con un ritardo pari a 2τ; come prima, tale nuova condizione persiste anche se S viene riportato a 0 (si veda figura 6.7b), X e Y rimangono nella stessa configurazione acquisita, cioè $X = 1$ e $Y = 0$, poiché essa è stabile rispetto a $R = S = 0$, a norma della tabella 6.5b. E' chiaro che ora, se anche S venisse riportato a 1, non cambierebbe

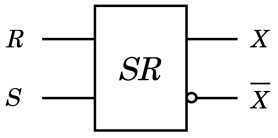
più nulla, perchè con $X = 1$ e $Y = 0$, S può stare stabilmente in ciascuno dei due stati. Quello che è successo è che inviando un impulso sull'ingresso S, che viene chiamato di impulso di *Set*, l'ingresso X va (o permane) a 1. Il

R	S	x	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	–
1	1	1	–

(a) Tavola di verità con lo stato futuro



(b) Mappa di Karnaugh della funzione X di figura 6.8a



(c) Simbolo circuitale di un *flip-flop* SR

Figura 6.8: Tavola di verità e mappa di Karnaugh per lo stato futuro

passaggio di stato della X viene evidenziato in figura 6.8a, nella quale si distingue tra stato corrente, indicato con la lettera x, e stato futuro, che viene invece indicato con X. Ricapitolando, il funzionamento di questo dispositivo può essere descritto nel modo seguente: quando $R = S = 0$, un impulso su S porta l'uscita X a 1; un impulso su R riporta l'uscita X a 0; ciò viene evidenziato in figura 6.8a dalle cifre in blu (*Set*) e in rosso (*Reset*). Nella figura 6.8b viene evidenziata la mappa di Karnaugh della funzione X, che consente di esprimere lo stato futuro in funzione di R, S e dello stato presente x

$$X = S + x\overline{R} \qquad \text{col vincolo che} \qquad R\,S = 0 \tag{6.2}$$

Un *flip-flop* che operi nel modo descritto prende il nome di *Flip-Flop Set-Reset*, e viene indicato con il simbolo SR. La ragione per cui i due ingressi non possono essere contemporaneamente 1 è duplice; in primo luogo perché essi porterebbero ambedue le uscite a 0, violando la condizione base di funzionamento di un *flip-flop*, secondo la quale le due uscite devono essere sempre complementari; in secondo luogo, se ambedue gli ingressi tornassero a 0 al medesimo istante, lo stato in cui il *flip-flop* si porterebbe non sarebbe prevedibile e al limite potrebbe realizzarsi una condizione di oscillazione. Con il vincolo $R\,S = 0$ il *flip-flop* SR realizza invece un dispositivo di memorizzazione affidabile, in cui lo stato delle uscite indica quale dei due ingressi si è trovato per ultimo al livello 1.
Poiché un tale circuito reagisce immediatamente alle variazioni dell'ingresso (a parte i tempi di commutazione), portandosi nello stato futuro previsto dalla funzione di transizione di stato, il suo funzionamento è di tipo *asincrono*, cioè non legato ad alcuna forma di sincronismo. E' inoltre evidente che l'impulso d'ingresso che fa commutare il dispositivo deve avere una durata minima se si vuole che la commutazione avvenga con sicurezza; si supponga infatti che S ritorni a zero prima che sia passato il tempo τ; in tal caso la variazione della variabile S non si propaga in uscita e la X rimarrebbe a 0. Il verificarsi di tale condizione renderebbe evidentemente incerto il funzionamento del circuito.

Latch di NAND

Un discorso del tutto analogo potrebbe essere fatto ricorrendo a due porte NAND al posto delle porte NOR, così come rappresentato in figura 6.9.
Le equazioni del sistema sono

$$\begin{aligned} X &= \overline{RY} \qquad Y = \overline{SX} \qquad \text{e sostituendo} \\ X &= \overline{R \cdot \overline{SX}} \\ &= \overline{R} + \overline{\overline{SX}} \qquad \text{De Morgan} \\ &= \overline{R} + SX \end{aligned} \tag{6.3}$$

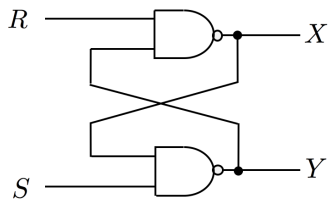


Figura 6.9: Flip-flop realizzato con due porte NAND

e appaiono come le duali delle 6.1. Se costruiamo le tavole di verità analoghe a quelle di figura 6.5 otteniamo le tavole di figura 6.10, dalle quali si deduce che ora la condizione da evitare è $R = S = 0$. Con questa ipotesi si ha sempre $Y = \overline{X}$, come si richiede a un *Flip Flop* SR. Rifacendo lo stesso ragionamento fatto sulle porte NOR si

R	S	X	Y	$\overline{R} + SX$	$\overline{SX}$
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	1	0

(a) Tavola di verità dell'equazione 6.3 per tutti gli stati possibili

R	S	X	Y	$\overline{R} + SX$	$\overline{SX}$
0	0	1	1	1	1
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	1	0	1
1	1	1	0	1	0

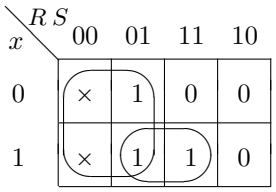
(b) Tavola di verità dell'equazione 6.3 riferita ai soli stati stabili, cioè quelli per i quali $X = \overline{R} + SX$ e $Y = \overline{SX}$

Figura 6.10: Tavola di verità dell'equazione 6.3

osserva che ora lo stato di partenza è a livello alto ($R = 1$, $S = 1$), mentre i segnali di attivazione sono a livello basso ($R = 0$ oppure $S = 0$). Ciò è coerente con la dualità tra le relazioni 6.1 e 6.3 e implica che dare un segnale di *Set* significa porre $S = 0$, ottenendo che X vada o permanga a 0; viceversa dare un segnale di *Reset* significa porre $R = 0$, ottenendo che X vada o permanga a 1. Questo comportamento è coerente con la tavola 6.10b.

R	S	x	X
0	0	0	–
0	0	1	–
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a) Tavola di verità con lo stato futuro



(b) Mappa di Karnaugh della funzione X di figura 6.11a

Figura 6.11: Tavola di verità e mappa di Karnaugh per lo stato futuro

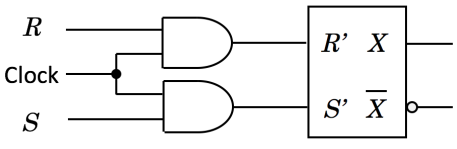
Il passaggio di stato della X viene evidenziato in figura 6.11a, analoga alla tavola 6.8a. Ricapitolando, il funzionamento di questo dispositivo può essere descritto nel modo seguente: quando $R = S = 1$, un impulso (negativo) su S porta l'uscita X a 0; un impulso (negativo) su R riporta l'uscita X a 1; ciò viene evidenziato in figura 6.11a dalle cifre in blu (*Set*) e in rosso (*Reset*). Nella figura 6.11b viene evidenziata la mappa di Karnaugh della funzione X, che consente di esprimere lo stato futuro in funzione di R, S e dello stato presente x

$$X = \overline{R} + xS \qquad \text{col vincolo che} \qquad R + S = 1 \tag{6.4}$$

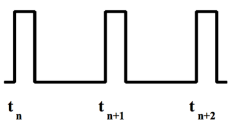
6.3 Moduli sequenziali sincroni

6.3.1 Il *Flip-Flop* SR sincrono

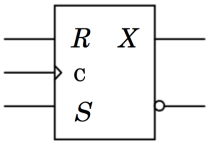
Il progetto di un circuito sequenziale può essere notevolmente semplificato se le commutazioni possono avvenire solo in corrispondenza in precisi istanti di tempo equintervallati. Questo tipo di funzionamento può esser assicurato se tutti i cambiamenti di stato vengono sincronizzati da opportuni impulsi di orologio (*clock*). E' bene chiarire che per *impulso* s'intende un segnale che normalmente si mantiene a un livello, usualmente 0, e va all'altro livello solamente per intervalli di tempo estremamente brevi. Viene invece chiamato *a livelli* un segnale che può rimanere sia a 0 che a 1 per periodi di tempo indefiniti e comunque molto lunghi se paragonati alla durata di un impulso. La dizione "estremamente breve" va evidentemente rapportata alla velocità del circuito, ma normalmente indica una durata dello stesso ordine di grandezza del tempo di ritardo del *flip-flop*. Dunque un circuito sequenziale



(a) *Flip-Flop* SR sincronizzato da un impulso di *clock*



(b) Impulsi di *clock*



(c) Simbolo di un *Flip-Flop* SR sincrono

Figura 6.12: *Flip-Flop* SR sincrono e relativo impulso di sincronizzazione

sincronizzato da un impulso di *clock*, detto circuito *sincrono* o circuito *clock-mode*, può cambiare stato solo in corrispondenza di tale impulso; inoltre potrà cambiare stato non più di una volta per ciascun impulso di *clock*. L'unico vincolo necessario per il corretto funzionamento del circuito è che i segnali S e R non cambino durante l'intera durata dell'impulso di clock.

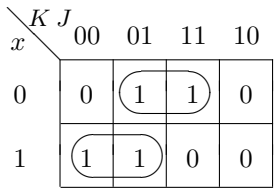
Per realizzare FFSR sincrono bisogna creare un consenso con delle porte AND, così come illustrato in figura 6.12a; il funzionamento del circuito è evidente: quando il segnale di *clock* di figura 6.12b ha valore basso, si ha $C = 0$ e dunque $S' = 0$ e $R' = 0$ indipendentemente dai valori di S e R. Quando invece l'impulso di *clock* scatta, si ha $C = 1$, le porte AND diventano trasparenti rispetto a S e R e il *flip-flop* funziona al solito modo.
La tavola di verità e le equazioni che reggono il funzionamento del *flip-flop* sincrono sono le stesse del *flip-flop* SR asincrono; in questo caso S, R e x indicano i valori presenti durante l'impulso di clock, mentre X si riferisce al valore assunto dall'uscita immediatamente dopo tale impulso.

6.3.2 Il *Flip-Flop* JK - FFJK

Malgrado il *flip-flop* SR sincrono sia assolutamente adatto a qualsiasi realizzazione circuitale, esso è raramente impiegato in pratica, poiché si preferisce far uso del *flip-flop* JK (FFJK) che ora descriviamo. In quest'ultimo il vincolo che ambedue gli ingressi non possano essere 1 allo stesso istante viene a cadere. La tavola di verità del *flip-flop* JK e la corrispondente mappa di Karnaugh sono riportate in figura 6.14. Si noti che il *flip-flop* JK è identico a quello RS con J corrispondente a S e K a R eccetto quando ambedue gli ingressi valgono 1, caso in cui il flip-flop comunque cambia stato. Dalla mappa di Karnaugh si evidenzia la struttura della funzione X

K	J	x	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	**0**	**1**
1	1	**1**	**0**

(a) Tavola di verità con lo stato futuro di un *flip-flop* JK

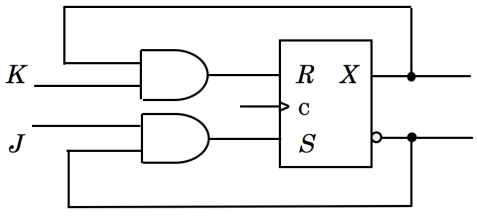


(b) Mappa di Karnaugh della funzione X di figura 6.13a

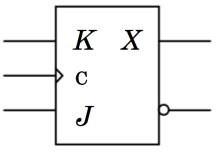
Figura 6.13: Tavola di verità di un *Flip-Flop* JK e semplificazione sulla mappa di Karnaugh

$$X = \overline{x}J + x\overline{K} \tag{6.5}$$

La realizzazione pratica di un FFJK si attua con l'aggiunta di due porte AND a un FFRS sincrono, così come evidenziato in figura 6.14a: quando $x = 1$ si abilita la sola porta AND del *reset* K, quando $x = 0$ si abilita la sola porta AND del *set* J; se entrambi J e K sono a 1, $x = 1$ forza un *reset*, mentre $x = 0$ forza un *set*.



(a) *Flip-Flop* JK realizzato a partire da un RS sincronizzato



(b) Simbolo circuitale di un *flip-flop* JK

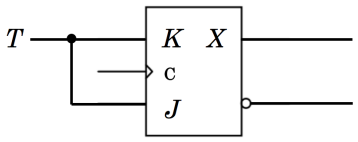
Figura 6.14: *Flip-Flop* JK

6.3.3 *Flip-Flop* di tipo T e D

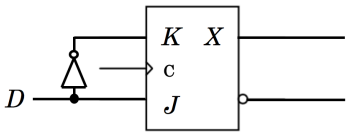
Un ulteriore tipo di *flip-flop* è il *flip-flop* T (FFT) - dove la T sta per *Toggle* - in cui il *clock* è il solo segnale d'ingresso, che in questo caso viene chiamato segnale di *trigger*. Quando l'ingresso T è a 1, a ogni istante di *clock* il *flip-flop* cambia stato; se invece l'ingresso T è a 0 esso rimane nello stesso stato; l'equazione di funzionamento si ricava dalla 6.5 ponendo $J = T$ e $K = T$:

$$X = \overline{x}T + x\overline{T} = \begin{cases} \overline{x} & \text{se } T = 1 \\ x & \text{se } T = 0 \end{cases}$$

Come evidenziato dalla figura 6.15a, un FFT si ottiene direttamente da un FFJK semplicemente connettendo assieme gli ingressi J e K; in questo modo, ogniqualvolta si ha $T = 1$, entrambi J e K sono a 1 e si ha la commutazione dello stato finale per quanto detto precedentemente.



(a) *Flip-Flop* T realizzato a partire da un FFJK



(b) *Flip-Flop* D realizzato a partire da un FFJK

Figura 6.15: *Flip-Flop* del tipo T e D

Un impiego tipico del FFT è quello di dimezzare la frequenza di *clock*, poiché una volta cambiato lo stato di uscita esso non cambia più fino al prossimo impulso di *clock*.
L'ultimo *flip-flop* che analizziamo è quello di tipo D (FFD) - dove la D sta per *Delay*. In esso l'uscita dopo un impulso di *clock* è uguale al valore presente all'ingresso D all'istante di clock. L'equazione di funzionamento si ricava dalla 6.5 ponendo $J = D$ e $K = \overline{D}$

$$X = \overline{x}D + x\overline{\overline{D}} = D \tag{6.6}$$

Dunque lo stato futuro X è lo stato che si avrà nel prossimo impulso di clock a seguito dell'ingresso D. In tal modo il *flip-flop* D realizza il trasferimento in uscita del segnale d'ingresso D con un ritardo pari al periodo di clock T.

6.4 Registri e contatori

I *flip-flop* costituiscono la circuiteria di base per la memorizzazione di singoli bit in formato elettronico. A partire da essi si possono costruire unità per la memorizzazione di blocchi di m bit denominati *registri*; la figura 6.16 ci mostra un esempio in tal senso, nel quale si costruisce un registro di m celle di memoria a partire da m *flip-flop* di tipo D.

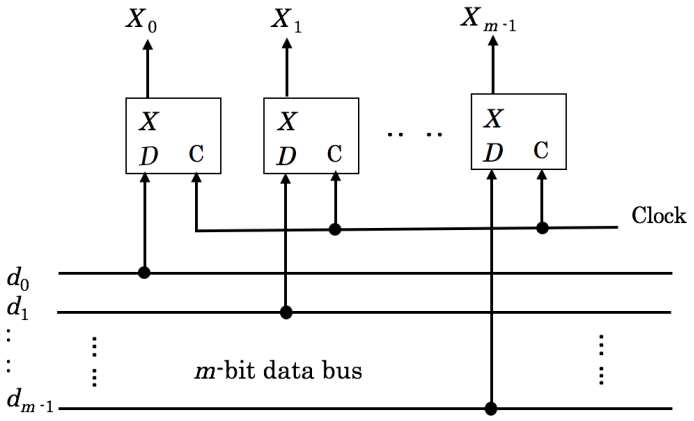


Figura 6.16: Registro di memoria da m-bit

Gli stessi m *flip-flop* di tipo D possono essere organizzati per realizzare i *registri a scorrimento*, che sono di fondamentale importanza per i flussi informativi interni ai calcolatori. Il circuito è rappresentato in figura 6.17 e il suo funzionamento è intuitivo: a ogni istante di tempo il contenuto del registro j-esimo si sposta nel registro $(j-1)$-esimo e si rende disponibile per l'uscita.

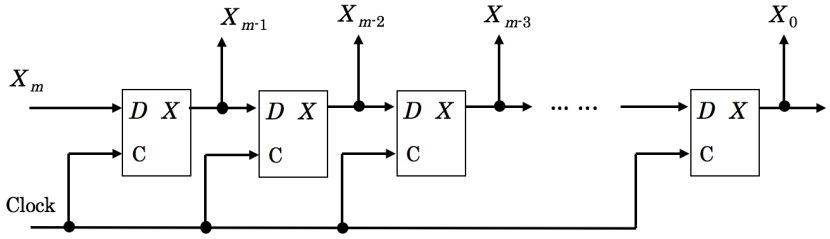


Figura 6.17: Registro a scorrimento

Con m *flip-flop* di tipo T è possibile realizzare anche un *contatore*, che scandisce, una dopo l'altra, tutte le 2^m configurazioni da $00...0$ a $111...1$. Per capirne il funzionamento facciamo riferimento al contatore a 2 bit di figura 6.18, partendo dalla configurazione 00. Il *flip-flop* di sinistra ha sempre $J = K = 1$, e in corrispondenza del primo

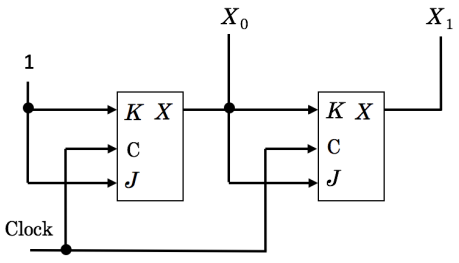


Figura 6.18: Contatore a 2 bit che passa attraverso gli stati $00, 10, 01, 11$ per le variabili X_0X_1

istante di *clock* cambia lo stato da $X_0 = 0$ a $X_0 = 1$; al successivo si ha $X_0 = 0$ e così via, per ogni istante in cui compare il segnale di *clock*. Prima che avvenga la commutazione si ha $X_0 = 0$, e dunque J=K=0 per il *flip-flop*

di destra; dalla tavola di verità 6.13a di un *flip-flop* JK si vede che, quando $J{=}K{=}0$ lo stato finale della variabile X viene conservato, e dunque nel momento in cui il *clock* si manifesta, mentre X_0 cambia X_1 resta nello stato iniziale, cioè $X_1 = 0$; il nuovo stato successivo a 00 è dunque 10. In queste condizioni all'avvento del secondo *clock* X_0 cambia nuovamente e si riporta a 0, mentre sull'ingresso del *flip-flop* di destra si ha ora $J{=}K{=}1$, che fa commutare lo stato finale X_1 da 0 a 1; il terzo stato è dunque 01. Nell'ultimo passo c'è il nuovo cambio di X_0, che torna a 1, con $J{=}K{=}0$ nel *flip-flop* di destra, che non porta variazioni all'uscita X_1, che rimane quindi a 1; il quarto stato è dunque 11. A seguito di un nuovo segnale di clock X_0 torna a 0 e X_1 cambia stato, riportandosi a 0. E il ciclo ricomincia. Il contatore può essere facilmente esteso a 3 bit introducendo una terza variabile X_2

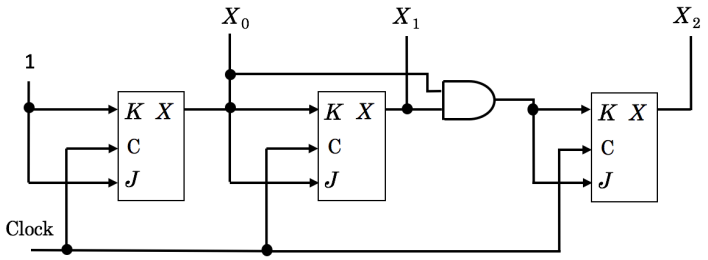


Figura 6.19: Contatore a 3 bit che passa attraverso gli stati $000, 100, 010, 110, 001, 101, 011, 111$ per le variabili $X_0X_1X_2$

anche senza fare un progetto sistematico basato sulle tavole di verità e sulle mappe di Karnaugh; a tal scopo basta considerare che per far commutare X_2 da 0 a 1 è necessario che sia $X_0 = X_1 = 1$. Quando ciò avviene X_2 rimane a 1, finché non si ripresenta nuovamente la coppia $X_0 = X_1 = 1$; a quel punto X_2 torna a 0. Tutto ciò viene realizzato con una porta AND, che consente di costruire il contatore a 3 bit di figura 6.19.

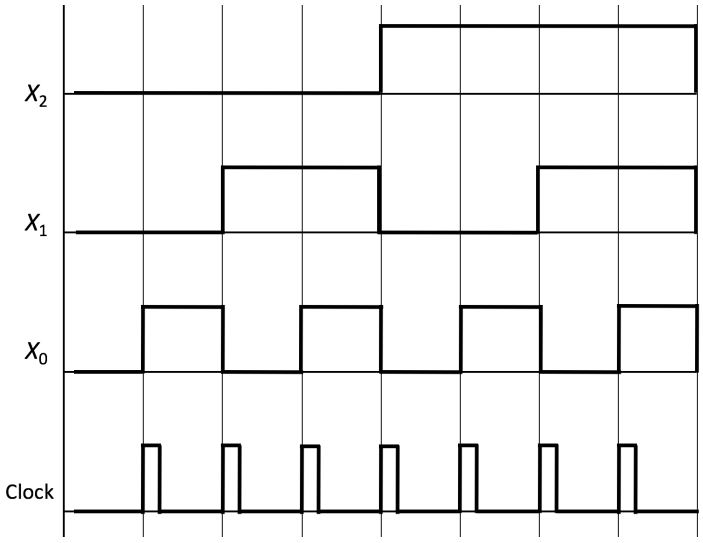


Figura 6.20: Andamento dei segnali associati alle variabili $X_0X_1X_2$ del circuito di figura 6.19, le cui variazioni sono controllate dall'impulso di *Clock*

Capitolo 7

L'architettura dei calcolatori

7.1 L'architettura di von Neumann

È a tutti noto che i calcolatori sono *programmabili*, e dobbiamo a Charles Babbage l'idea di poter risolvere problemi diversi anche senza modificare l'*hardware* del sistema, e ciò proprio grazie a una riprogrammazione dello stesso.
Un programma $\mathcal{P}$ è genericamente costituito da un insieme di *istruzioni* $I(1), I(2), \ldots, I(n)$, che devono essere eseguite in sequenza, partendo cioè dall'esecuzione di $I(1)$, cui seguirà quella di $I(2)$ e così via, finché o sono state eseguite tutte le istruzioni o si è pervenuti a un blocco della computazione. Questo è perlomeno l'approccio seguito nell'ambito della *programmazione imperativa*, la quale è un paradigma di programmazione secondo cui un programma viene appunto inteso come un insieme di istruzioni (o comandi), ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato. Da un punto di vista sintattico, i costrutti di un linguaggio imperativo sono spesso identificati da verbi all'imperativo, quali per esempio *print, read, copy,....*
Nelle prime realizzazioni di computer (Z1, Colossus, ENIAC), il programma $\mathcal{P}$ era esterno alla macchina o addirittura cablato, ma le limitazioni di tale impostazione portarono ben presto *John von Neumann* a proporre un modello architetturale di *computer a programma memorizzato*. Sulla base di questa nuova impostazione vennero poi realizzati i primi calcolatori con architettura di von Neumann, quali il SSEM (*Manchester Small-Scale Experimental Machine*), l'EDSAC (*Electronic Delay Storage Automatic Calculator*) e l'EDVAC (*Electronic Discrete Variable Automatic Computer*). Un tale approccio si poté attuare soprattutto grazie allo sviluppo delle memorie a ferrite (cfr. fig.2.24c), introdotte nei primi anni '50.
L'essenza di un computer a programma memorizzato è allora costituita da:

1. una *memoria indirizzabile*, che possa contenere il programma e i dati
2. un'*unità logico-aritmetica*, che possa lavorare sui dati della memoria
3. un *program counter*, cioè un registro che indica l'indirizzo di memoria dell'istruzione che deve essere eseguita

Se dunque il programma $\mathcal{P}$, costituito dalle istruzioni $I(1), I(2), \ldots, I(n)$, deve essere eseguito, si dovrà partire dall'esecuzione di $I(1)$, cui seguirà quella di $I(2)$ e così via. Se $\mathcal{P}$ è contenuto nella memoria del calcolatore, bisogna sapere *dove* reperire $I(1)$, cioè quale sia *l'indirizzo* di $I(1)$ all'interno della memoria. Questa informazione

viene memorizzata su un registro chiamato *program counter*. Quando l'istruzione $I(1)$ è stata individuata, bisogna acquisirla (*fetch*) e metterla a disposizione dell'unità logico-aritmetica, decodificarla (*decode*) e successivamente eseguirla (*execute*). Questa sequenza di operazioni prende il nome di *ciclo fetch-decode-execute* del processore. Eseguita l'istruzione $I(1)$ il *program counter* incrementerà di 1 il suo valore, punterà all'indirizzo dell'istruzione successiva $I(2)$ e il processore attiverà un nuovo ciclo *fetch-decode-execute* per l'esecuzione di $I(2)$. Dal punto di vista logico un computer programmabile svolge, per ogni programma $\mathcal{P}$, il seguente semplice ciclo:

```
PC ← 0;
repeat
    istruzione ← memoria[PC];
    decode(istruzione);
    fetch(operandi);
    execute
    PC ← PC+1
until istruzione = STOP
```

Vediamo ora nel dettaglio il significato di queste operazioni:

PC $\leftarrow 0$ significa che il registro PC, il *program counter*, viene caricata col valore 0, che per ipotesi corrisponde all'indirizzo della cella di memoria che contiene la prima istruzione che deve essere eseguita.

Il ciclo **repeat** I **until** C (ciclo *repeat*), dove C è una *condizione* e I una generica istruzione elementare (o un sottoprogramma costituito da istruzioni elementari, come nel nostro caso) significa:

ripeti (*repeat*) l'istruzione I, finché (*until*) si realizza la condizione C

Nel nostro caso la condizione C che ferma l'esecuzione è che l'istruzione corrente sia uno STOP.

istruzione ← *memoria*[PC] significa che la prima istruzione di $\mathcal{P}$, memorizzata nella postazione di memoria indicata dal PC, viene caricata su un registro opportuno di memoria e messa a disposizione dell'unità logico-aritmetica del processore.

decode(istruzione) significa che il processore provvede a decodificare il tipo di istruzione che deve essere eseguita.

fetch(operandi) significa che si raccolgono gli operandi sui quali interviene l'istruzione.

execute significa che il processore esegue l'istruzione sui propri operandi.

PC ← PC+1 significa il valore del *program counter* viene incrementato di 1.

A questo punto si controlla la condizione C del ciclo *repeat*, cioè se l'istruzione corrente appena eseguita è un'istruzione di STOP. In tal caso si termina la computazione, altrimenti si rientra nel ciclo *repeat* per eseguire $I(2)$.

L'architettura di von Neumann per un computer a programma memorizzato prevede in prima approssimazione i seguenti elementi (si veda figura 7.1):

RAM - *Random Access Memory* - una memoria indirizzabile che possa contenere il programma e i dati;

PC - *Program Counter* - un registro di memoria che indica l'indirizzo della cella di memoria che contiene l'istruzione che deve essere eseguita;

RI - *Registro Istruzioni* - registro sul quale viene caricata l'istruzione che deve essere eseguita;

ACC - *Accumulator* - registro sul quale viene caricato il risultato dell'elaborazione della ALU;

ALU - *Arithmetic Logic Unit* - unità logico-aritmetica che lavora sui dati della memoria;

CPU - *Central Processing Unit* - è il processore e costituisce il cuore del sistema; essa contiene la ALU e i registri necessari a memorizzare le istruzioni che devono essere eseguite e i risultati intermedi delle operazioni;

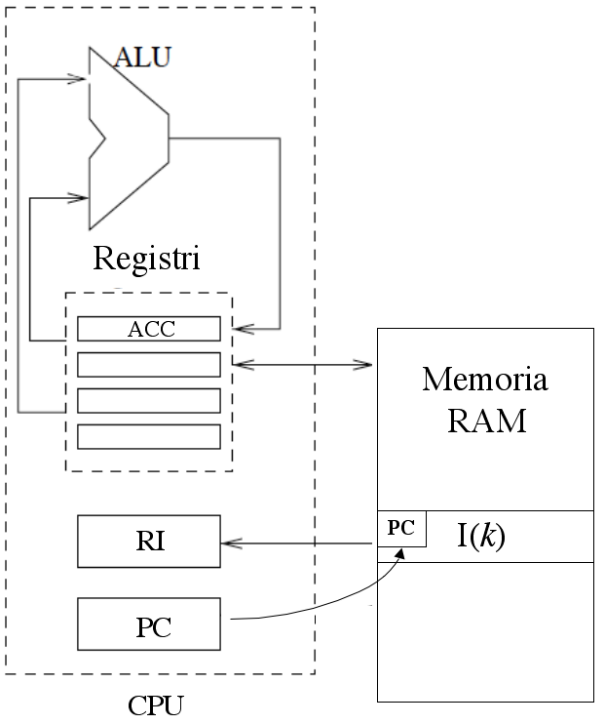


Figura 7.1: Architettura di von Neumann di un calcolatore a programma memorizzato

Nella struttura architetturale appena delineata la memoria RAM, detta anche *memoria principale*, contiene il programma e i dati intermedi; lavorando in stretta connessione col processore essa dovrà essere particolarmente veloce ed efficiente. D'altra parte quando il programma non serve esso deve risiedere da qualche parte, senza dover necessariamente occupare lo spazio della memoria RAM, che per le caratteristiche richieste sarà presumibilmente costosa, e di conseguenza non troppo capiente. È allora necessario aggiungere al sistema una memoria più esterna, chiamata *memoria secondaria* o *memoria di massa*, sulla quale potremo conservare tutti i programmi in uso all'utente dopo averli installati. Ecco allora che quando si attiva un'icona col doppio *clic* per aprire p.es. un file di testo, quello che accade è che il programma di elaborazione testi, inizialmente memorizzato sulla memoria di massa, viene caricato sulla memoria RAM e messo a disposizione del processore per la sua esecuzione, che in questo caso consisterà nella gestione e nella modifica del testo. L'interfaccia del computer con il mondo esterno passa attraverso le *periferiche*, che costituiscono le unità di ingresso e uscita dell'informazione; alcuni esempi tipici di periferiche sono: la tastiera, il *mouse*, un'unità di conversione A/D, per quanto riguarda i dati in ingresso; lo schermo e una memoria esterna per quanto riguarda dati in uscita. Un esempio della struttura completa di un calcolatore è fornito in figura 7.2.

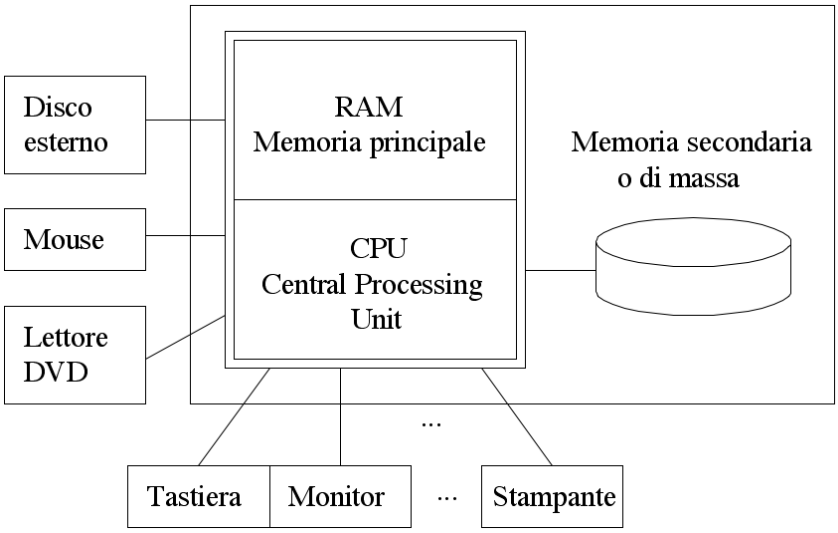


Figura 7.2: Struttura schematica di un calcolatore

Possiamo allora affermare che un computer è costituito sostanzialmente dai seguenti quattro componenti principali:

1. Processore (CPU)
2. Memoria primaria (RAM)
3. Memoria secondaria o di massa
4. Periferiche

Analizziamo ora queste componenti in dettaglio.

7.1.1 Il processore (CPU)

Il processore è costituito dall'insieme dei circuiti e dei registri di memoria necessari per decodificare ed eseguire le istruzioni del programma. Il suo componente più importante è l'*Unità di Controllo* UC (fig.7.3), che contiene i circuiti logici necessari per coordinare e attuare tutte le operazioni necessarie per realizzare il ciclo *fetch-decode-execute*; in particolare essa scandisce, col suo orologio interno, tutta l'attività del processore, attiva il PC, coordina la fase di lettura dalla memoria RAM dell'istruzione da eseguire, la decodifica e la manda in esecuzione avvalendosi anche della ALU. Quest'ultima è supportata anche da un *processore matematico* per velocizzare le operazioni in virgola mobile; nei primi modelli di calcolatore esso era venduto a parte su richiesta e installato successivamente su uno zoccolo della scheda madre; oggi fa parte integrante del *chip* del processore. All'interno del processore si trova anche la batteria dei *registri* che costituiscono il primo livello di memoria, quello immediatamente disponibile all'*Unità di Controllo*. Abbiamo già anticipato la funzione del *Program Counter* PC, che contiene in ogni istante l'indirizzo della cella di memoria RAM con la prossima istruzione da eseguire. Quest'ultima viene poi caricata sul *Registro delle Istruzioni* RI, per essere decodificata dall'*Unità Centrale* e successivamente eseguita. Il *Registro di Stato* RS contiene un insieme di variabili booleane (*flag*) che specificano lo stato di alcune operazioni matematiche a seguito di test effettuati su condizioni richieste dai programmi. Come

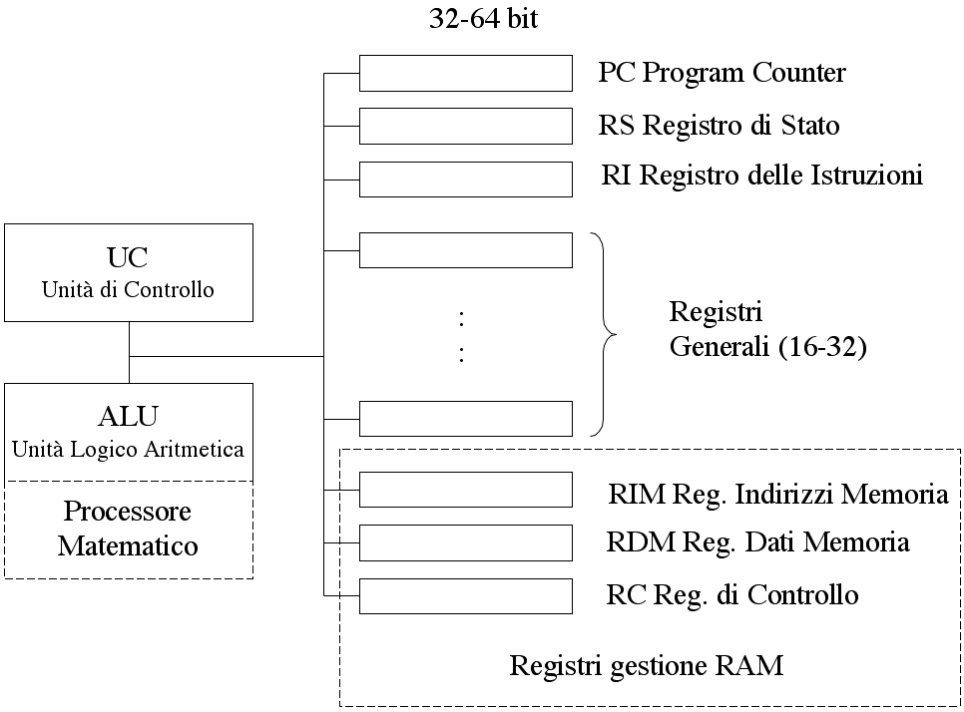


Figura 7.3: Struttura del processore

esempio di *flag* possiamo citare CF (*Carry flag*), che indica se il risultato di un'operazione produce un riporto non contenibile nei bit usati per il calcolo (nel caso di numeri *unsigned*), oppure OF (*Overflow*), che indica se il risultato di un'operazione è in *overflow* (nel caso di numeri *signed*, cioè con notazione in complemento a 2), secondo la rappresentazione in complemento a due. I *Registri Generali* sono invece impiegati per memorizzare i risultati intermedi delle operazioni (si veda un esempio in figura 7.4).

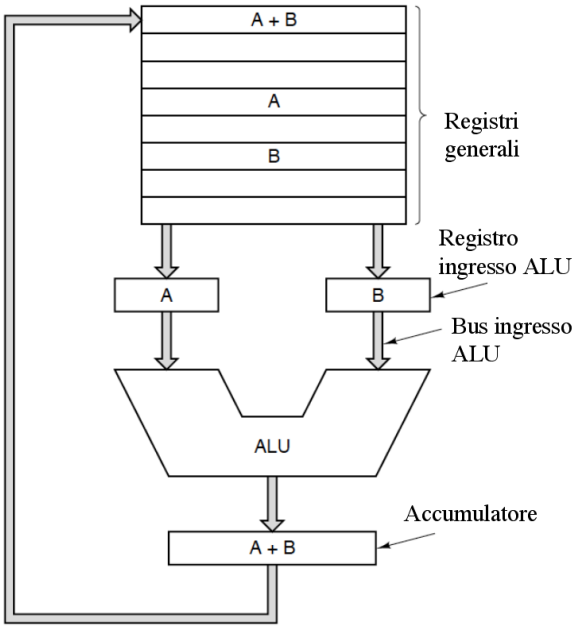
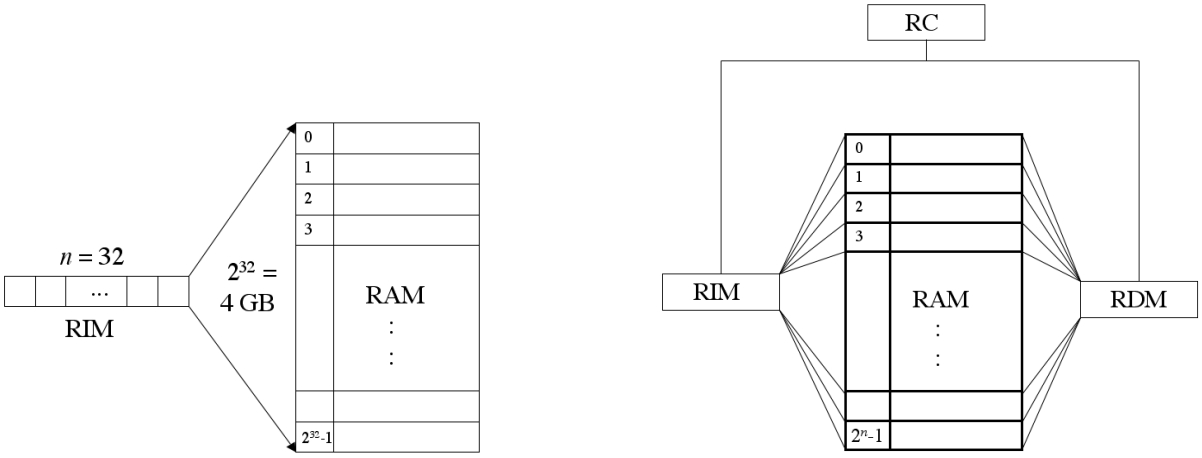


Figura 7.4: Esempio di operazione eseguita dalla ALU usando i Registri Generali

La schiera di registri viene completata dai registri necessari per gestire le operazioni lettura e la scrittura in memoria RAM. Essi sono il *Registro Indirizzi di Memoria* (RIM), sul quale viene caricato l'indirizzo della memoria interessato alla lettura o alla scrittura del dato; il *Registro Dati di Memoria* (RDM), che contiene il dato che deve essere scritto o letto e il *Registro di Controllo* (RC), che specifica se si tratta di lettura o scrittura.
Per poter accedere alla memoria RAM, il RIM deve indicare l'indirizzo della cella che si vuole usare in lettura o scrittura; se p.es. esso possiede 32 bit, allora saranno disponibili tutti gli indirizzi che vanno da 0 a $2^{32} - 1$; tale intervallo viene chiamato *spazio di indirizzamento* (si veda fig.7.5a).



(a) Spazio di indirizzamento ottenibile con 32 bit

(b) Gestione della lettura e della scrittura in memoria RAM

Figura 7.5: Spazio di indirizzamento del RIM e gestione delle operazioni in memoria RAM

Per sfruttare al massimo tutta l'estensione dell'intervallo, sarà opportuno avere una memoria con un numero

di celle esattamente pari allo spazio di indirizzamento (4 GB nell'esempio di cui sopra). Supponiamo ora che il dato debba essere letto dalla memoria; in tal caso il RC dà indicazione *read* e una volta individuata la cella con il RIM, il contenuto della stessa viene riversato nel RDM, che sarà poi a disposizione della UC. Se al contrario il dato deve essere memorizzato nella RAM, esso sarà già disponibile nel RDM, il RC darà indicazione *write* e il dato verrà caricato sulla cella con l'indirizzo specificato dal RIM (fig.7.5b).

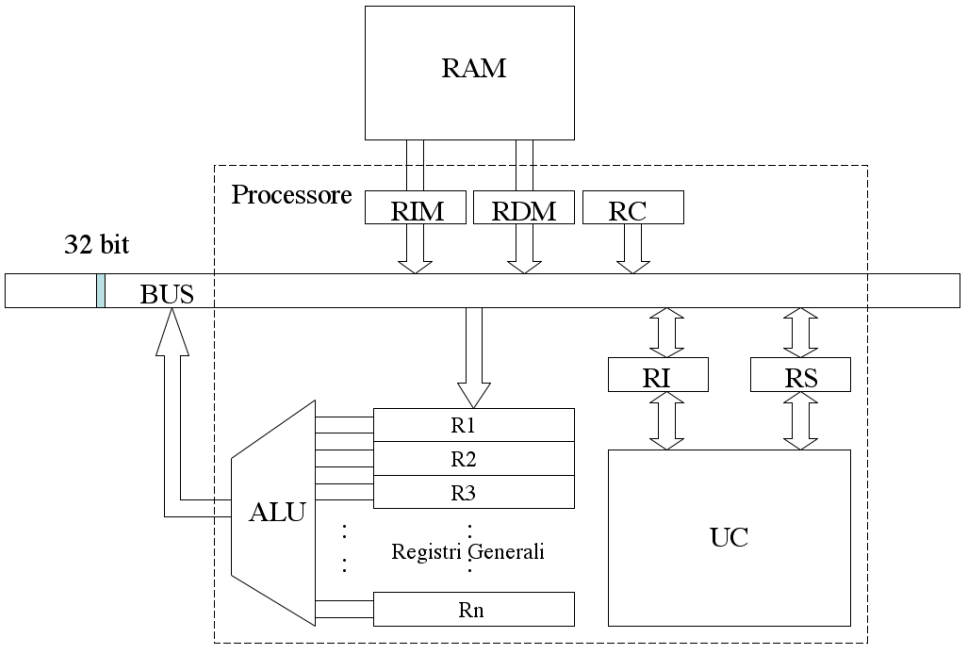


Figura 7.6: Il Bus dei dati mette in comunicazione le componenti di un processore e la memoria RAM

La successiva figura 7.6 mette in evidenza la linea di connessione parallela (*Bus*) che mette in comunicazione le componenti di un processore e la memoria RAM. L'insieme delle istruzioni che il calcolatore è in grado di eseguire tramite la CPU costituisce il *linguaggio macchina*; esse devono essere molto semplici e direttamente gestibili dai circuiti dell'unità di controllo e della ALU. Per le sue caratteristiche un tale linguaggio, molto legato alla struttura del processore, è inadatto alla programmazione da parte di operatori umani, ai quali si riserva invece l'impiego di linguaggi ad *alto livello* (come p.es. C, C++, *Java, Fortran, Pascal, Cobol*,...), le cui istruzioni elementari consentono di effettuare operazioni complesse e astratte, molto vicine alla logica che guida il ragionamento umano nell'approccio algoritmico-procedurale. In una prima fase storica molte architetture per computer cercarono di colmare lo iato semantico che esisteva tra i comandi in linguaggio macchina e quelli dei linguaggi ad alto livello. Questi calcolatori offrivano istruzioni che consentivano di eseguire operazioni relativamente complesse, quali la gestione delle procedure, la gestione dei *loop* e dei salti, la gestione di strutture dati in memoria e altri compiti comuni. Un tale approccio, noto con l'acronimo CISC (*Complex Instruction Set Computer*), permetteva la realizzazione di programmi compatti e che quindi richiedevano poca memoria, una risorsa molto costosa negli anni '60. In seguito prese però forza anche la filosofia di progettazione chiamata RISC (*Reduced Instruction Set Computer*), basata su un numero ridotto di istruzioni in linguaggio macchina. Per esempio le architetture RISC permettono di accedere alla memoria unicamente tramite delle istruzioni specifiche (*load* e *store*) che provvedono a leggere e scrivere i dati nei registri del microprocessore, mentre tutte le altre istruzioni manipolano i dati contenuti all'interno del processore. Nei processori CISC vale l'esatto opposto, poiché praticamente tutte le istruzioni possono accedere ai registri o alla memoria in modo indifferente.
Tralasciando il problema troppo specifico della differenza tra CISC e RISC, che fra l'altro sta perdendo d'importanza con le nuove generazioni di processori paralleli, possiamo dire che le istruzioni in linguaggio macchina di un processore sono caratterizzate da un *codice operativo*, che specifica il tipo di istruzione, e da 2 o 3 operandi, che

specificano i registri che contengono i dati sui quali si vuole eseguire l'istruzione. Esse appartengono alle seguenti categorie:

Istruzioni di I/O - servono per portare i dati all'interno o all'esterno della CPU; prevedono l'indicazione della periferica interessata e della locazione di memoria dove si trova il dato da portare in uscita o dove si vuole immettere il dato proveniente dall'esterno

READ INP 1322 Legge da tastiera e mette il dato nella cella di indirizzo 1322 della memoria RAM

WRITE OUT 1902 Scrive a video il contenuto della cella di indirizzo 1902 della memoria RAM

Istruzioni logico-aritmetiche - sono le istruzioni di manipolazione logico-aritmetica dei dati; devono essere indicati i dati su cui operare e la destinazione del risultato;

ADD R1 R2 Addiziona il contenuto del registro R1 con il contenuto del registro R2

MULT R1 1312 Moltiplica il contenuto della cella di indirizzo 1312 della memoria RAM con il contenuto del registro R2

COMP R1 R3 RC Compara il contenuto dei due registri R1 e R3 e scrive -1, 0 o 1 in RC a seconda che sia $R1 > R3$, $R1 = R3$, $R1 < R3$

Istruzioni di accesso alla memoria - servono a spostare dati da locazioni di memoria a registri della CPU o viceversa

LOAD 1672 R2 Carica sul registro R2 il contenuto della cella RAM d'indirizzo 1672

STORE R1 1559 Memorizza il contenuto del registro R1 nella cella RAM d'indirizzo 1559

Istruzioni di salto - servono a modificare la sequenza di esecuzione di un programma e devono contenere l'indirizzo di memoria dove si trova la prossima istruzione da eseguire

BRLT RC R1 1267 Se il contenuto del registro RC è minore del contenuto del registro R1 salta all'istruzione contenuta nella cella 1267, altrimenti incrementa PC di 1

JUMP 1721 Salta all'istruzione contenuta nella cella 1721

Siamo ora in grado di descrivere un ciclo completo *fetch-decode-execute* del processore. Supponiamo che il programma da eseguire sia contenuto nella memoria RAM a partire dalla cella di indirizzo 1000 e che le prime due istruzioni siano:

Load 1915 R1
Add R1 R2
$\vdots$

Con riferimento alla figura 7.7 elenchiamo i passi del ciclo *fetch-decode-execute* necessari per eseguire la prima istruzione Load 1915 R1:

1. poiché l'istruzione si trova nella cella di indirizzo 1000 della RAM, è necessario che questo valore venga passato ai registri che si occupano della lettura/scrittura in memoria; il valore 1000 viene quindi copiato nel registro RIM
2. il registro di controllo RC dispone la lettura della memoria con il comando *Read*

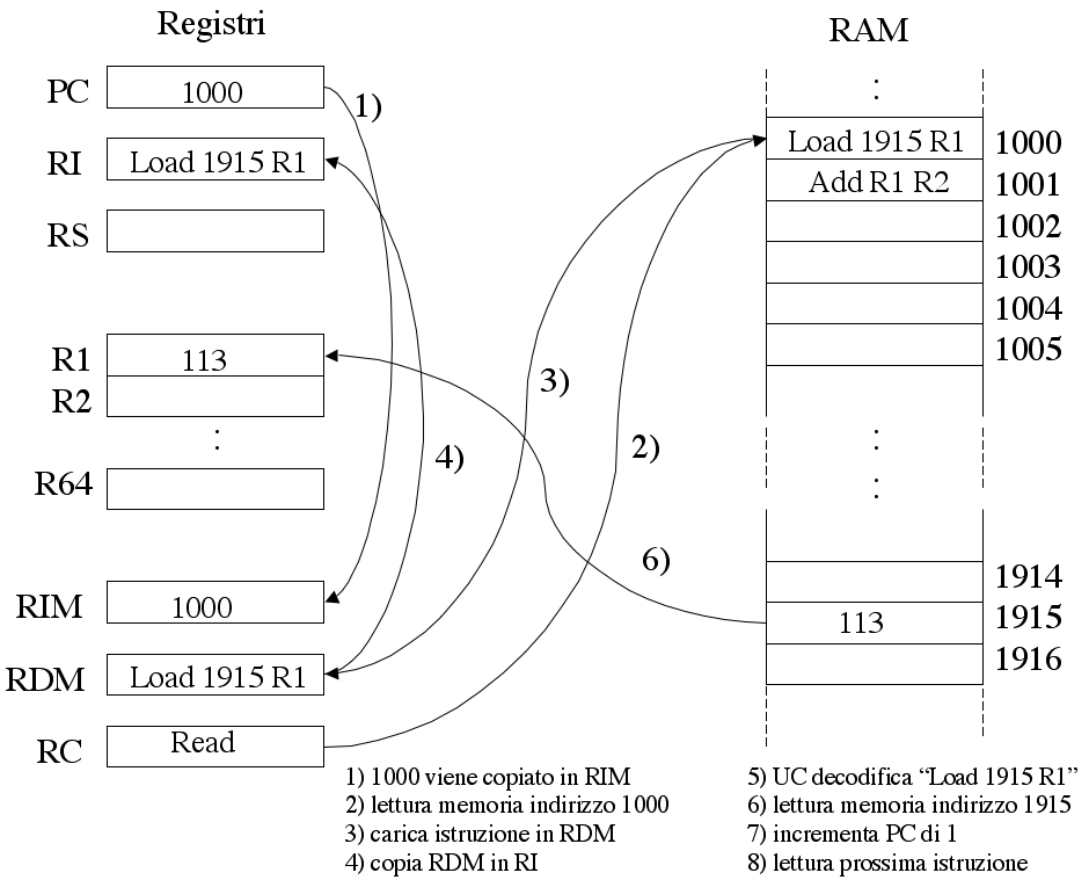


Figura 7.7: Ciclo *fetch-decode-execute* per l'esecuzione dell'istruzione Load 1915 R1

3. il contenuto della memoria all'indirizzo 1000, costituito dall'istruzione da mandare in esecuzione, viene caricato nel registro RDM

4. il contenuto di RDM può ora essere posto a disposizione della UC, caricando l'istruzione che deve essere posta in esecuzione nel registro delle istruzioni RI

5. UC decodifica l'istruzione "Load 1915 R1", che richiede il caricamento nel registro generale R1 del contenuto della cella 1915

6. viene attivato un nuovo ciclo di lettura della memoria RAM, per caricare il contenuto della cella 1915, cioè il valore 113, nel registro R1

7. a questo punto l'istruzione è stata eseguita e si deve passare all'istruzione successiva; il contenuto del PC viene incrementato di 1

8. si riparte con un nuovo ciclo *fetch-decode-execute* per la lettura della prossima istruzione

Volendo riassumere i caratteri essenziali dell'esecuzione di una qualunque istruzione possiamo dire che il procedimento richiede cinque *microistruzioni* fondamentali:

IF (Instruction Fetch) : Lettura dell'istruzione da memoria

ID (Instruction Decode) : Decodifica istruzione e lettura operandi dai registri

EX (Execution) : Esecuzione dell'istruzione

MEM (Memory) : Attivazione della memoria (solo per certe istruzioni)

WB (Write Back) : Scrittura del risultato nel registro opportuno

Ogni CPU in commercio è gestita da un orologio centrale (*clock*) che scandisce in tempo discreto il funzionamento del sistema, e ogni microistruzione richiede almeno un ciclo di *clock* per poter essere eseguita. Le prime CPU erano formate da un'unità polifunzionale che svolgeva in rigida sequenza tutti e cinque i passaggi legati all'elaborazione delle istruzioni. Una CPU classica richiedeva quindi almeno cinque cicli di *clock* per eseguire una singola istruzione. Nel seguito, con il progresso della miniaturizzazione su larga scala assicurata dalla legge di Moore (sez.1.5) è stato possibile integrare un numero maggiore di transistor in un microprocessore, consentendo di parallelizzare alcune operazioni, tra cui le cinque descritte poc'anzi. Nasce a questo punto il concetto di *pipeline* dei dati. Una CPU dotata di *pipeline* è composta da cinque linee parallele di elaborazione, ciascuna capace di eseguire sequenzialmente le microistruzioni sopra descritte; le linee lavorano sulla base di una traslazione temporale unitaria, in modo che quando la prima linea sta elaborando la seconda microistruzione ID della prima istruzione, la seconda linea elabora la prima microistruzione della seconda istruzione, e così via (fig.7.8). Quando la *pipeline* è a regime, dall'ultimo stadio esce ciclicamente a ogni istante di *clock* un'istruzione completata. Si guadagna quindi una maggior velocità di esecuzione a prezzo di una maggior complessità circuitale del microprocessore, che deve contenere cinque linee di elaborazione che lavorano in parallelo tra loro.

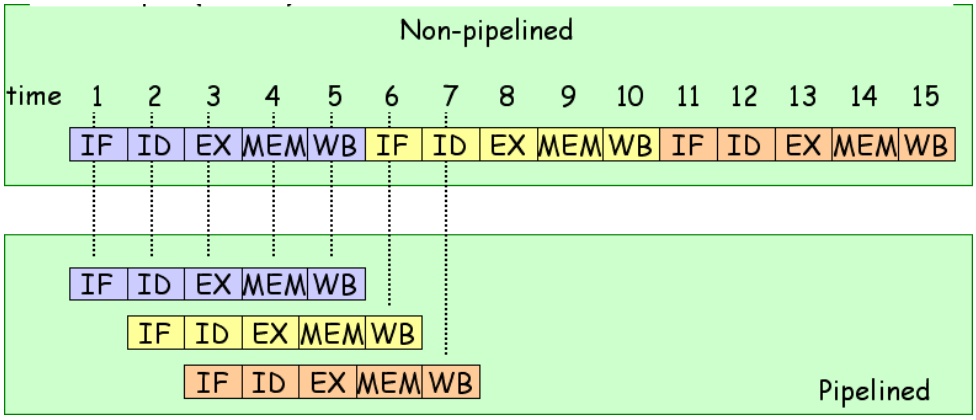


Figura 7.8: *Pipeline* dei dati per un microprocessore

7.1.2 La gerarchia delle macchine virtuali

L'esecuzione ripetuta del ciclo *fetch-decode-execute* consente l'esecuzione delle istruzioni che costituiscono i programmi che via via immettiamo nel calcolatore. Essa si realizza a un livello che è essenzialmente quello dei circuiti e delle memorie a registri del microprocessore. Si è più volte osservato che tali istruzioni sono estremamente semplici, quali caricare un dato in memoria, sommare il contenuto di due registri, leggere dalla memoria, fare un confronto tra due valori ecc. È evidente che progettare una procedura complessa per la soluzione di un problema della vita reale lavorando a questo livello di dettaglio per le istruzioni del programma sarebbe un'impresa (quasi) irrealizzabile. Per di più le istruzioni che abbiamo indicato come Load 1915 R1, COMP R1 R3 RC, JUMP 1721 ecc., descritte in un linguaggio chiamato *assembly*, non sarebbero comprensibili a livello circuitale, poichè come sappiamo il calcolatore opera col solo alfabeto binario; le vere istruzioni in *linguaggio macchina* si presentano in effetti come stringhe binarie; la figura 7.9 mostra un'ipotetica istruzione da 16 bit che compara il contenuto dei

due registri R1 e R3 e scrive un valore su un terzo registro RC. Nella figura viene evidenziata anche la stringa in

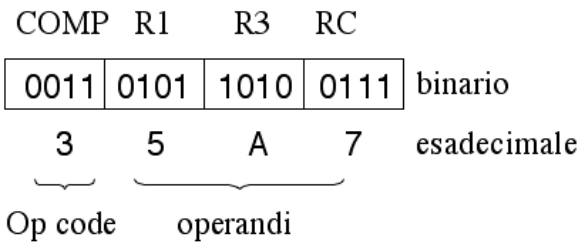


Figura 7.9: Istruzione in linguaggio *assembly* e corrispondente codifica binaria ed esadecimale

esadecimale, più compatta da trattare nel caso si voglia riprodurre in modo efficiente il listato del programma.
La programmazione basata sul linguaggio *assembly* segue le regole imposte dal corrispondente *Instruction Set Architecture* o *ISA*, che specifica l'insieme di tutte le istruzioni che possono essere attuate a questo livello; la traduzione delle istruzioni dal linguaggio *assembly* al linguaggio macchina viene attuata da un programma chiamato *assembler*. Tuttavia programmare in *assembly* non è efficiente, e i programmi che si realizzano sono vincolati alla tipologia architetturale della macchina in uso, e come tali non sono esportabili su altre macchine. È quindi necessario fare in modo che il programmatore possa operare a un livello logico superiore, usando linguaggi come C, C++, *Java, Fortran, Pascal*, ecc; per questi linguaggi le istruzioni elementari consentono di effettuare operazioni complesse e astratte, molto vicine alla logica che guida il ragionamento umano nell'approccio algoritmico-procedurale; inoltre i programmi in tali linguaggi sono indipendenti dall'architettura della macchina. La traduzione tra un linguaggio ad alto livello di questo tipo e il linguaggio *assembly* viene garantita da un programma che si chiama *compilatore*, che andrà adattato alle diverse piattaforme architetturali in uso.
Da quanto detto finora si intuisce che il *computer* è una struttura molto complessa, in grado di lavorare solo a un livello circuitale e binario. Per poter creare una connessione operativa con l'utente, che è un programmatore esterno, è allora necessario costruire una *gerarchia* di ambienti organizzata in *livelli di astrazione* o *macchine virtuali* i quali, partendo dai transistor in condizioni di saturazione o interdizione del livello circuitale più basso, possano giungere sino all'interfaccia grafica, basata sulle icone e sulla metafora della scrivania, che si presenta all'utente in procinto di programmare usando un linguaggio tra quelli prima menzionati. Ogni livello i, associato alla macchina virtuale M_i, è caratterizzato da un proprio linguaggio L_i che utilizza i comandi e i servizi messi a disposizione dal livello inferiore. A ogni livello superiore aumenta l'astrazione e la facilità d'uso, ma diminuisce la velocità di esecuzione. Per scrivere i programmi a un certo livello non è necessario conoscere come viene effettuata la traduzione e quindi l'esecuzione al livello inferiore. Nel passaggio tra un livello e l'altro si possono usare, in generale, due diversi approcci operativi distinti, quello basato sui *compilatori* e quello basato sugli *interpreti*.
Un *compilatore* traduce il programma $P(L_i)$, scritto in linguaggio L_i, in un programma $P(L_{i-1})$, scritto in linguaggio L_{i-1}. Il nuovo programma $P(L_{i-1})$ viene quindi eseguito in blocco.
Un *interprete* esamina il programma $P(L_i)$, scritto in linguaggio L_i e, istruzione per istruzione, lo traduce nel linguaggio L_{i-1} eseguendolo.
Se dunque chiamiamo M_0 la macchina virtuale cha abbiamo a disposizione al livello più basso e L_0 il corrispondente linguaggio, possiamo pensare di costruire una macchina virtuale M_1 con il relativo linguaggio L_1, che si pone a un livello astratto superiore e più vicino all'utente. Se seguiamo l'approccio del compilatore, possiamo eseguire un programma $P(L_1)$ nella macchina M_0 traducendo ogni istruzione di L_1 in una sequenza di istruzioni di L_0 ad essa equivalente. L'insieme delle istruzioni costituisce il programma $P(L_0)$ che può essere eseguito direttamente dalla macchina.
Un altro modo per eseguire un programma scritto nel linguaggio L_1 tramite le funzionalità di M_0 consiste nello scrivere un programma in L_0, detto appunto *interprete*, che sia in grado di eseguire *tutti i programmi in* L_1. Si osservi che la possibilità di costruire un interprete è un importante acquisizione teorica legata all'esistenza della *Macchina Universale*. Essa completa il percorso di astrazione delle funzioni delle macchine, che portò in un primo momento al passaggio dalle *macchine cablate* per la soluzione di un certo problema alle *macchine programmabili*, consentendo di risolvere problemi di natura diversa cambiando solo il programma della macchina e non la

sua struttura (si veda la sez.1.2). Successivamente il percorso si completò con la dimostrazione della possibilità di costruire un *programma universale*, che consente di risolvere qualunque problema (risolubile) usando la stessa struttura per la macchina e lo stesso *software*.
Tornando alla gerarchia dei livelli, osserviamo che per rendere traduzione e interpretazione utilizzabili in pratica, è opportuno che i linguaggi L_0 e L_1 non siano "troppo" diversi tra loro. La stessa idea si può applicare a L_2 e M_2, L_3 e M_3 ecc., dando luogo alla gerarchia che ora descriveremo brevemente a partire, per completezza, da un livello (-1) inferiore a quello d'interesse per l'architettura dei *computer*, cioè dal livello dei transistor.

Livello -1 - Elettronico Siamo al livello dei transistor che formano i circuiti elettronici che costituiscono le *porte logiche* di cui è composto un calcolatore (si vedano le figure 1.22). A questo livello il singolo transistor è in conduzione o in interdizione (si vedano le figure 2.6a e 2.6b), e si raggiunge un livello di dettaglio che viene in genere trascurato nella progettazione dei calcolatori.

Livello 0 - Logico Questo è il primo livello utile nella progettazione di un computer. La macchina è formata da porte logiche o *gate* studiate nei capitoli precedenti. Ogni porta riceve in ingresso dei segnali binari e calcola una semplice funzione Booleana (AND, OR, ...). Collegando opportunamente le porte di base si ottengono relazioni logiche complesse per i circuiti costituenti, p.es. si può realizzare una memoria di un bit (bistabile). Combinando n memorie di un bit si può formare un registro capace di memorizzare un numero binario compreso tra 0 e $2^n - 1$. Mediante le porte si realizzano i circuiti logici il cui funzionamento è regolato dalle leggi dell'algebra Booleana e dell'elettronica digitale. La macchina logica del livello 0 viene progettata dal costruttore dei vari componenti ed è puramente hardware.

Livello 1 - Microarchitettura A questo livello troviamo tutti gli elementi nell'architettura di base del computer, e cioè i registri generali usati come memoria locale, la ALU che esegue semplici operazioni logico-aritmetiche, gli elementi di connessione tra registri e ALU, i registri dedicati al controllo (PC, RI, ...) e la circuiteria dell'Unità di Controllo. Il percorso dei dati può essere gestito da un programma controllabile dall'esterno, chiamato *microprogramma* (come nel caso dell'architettura CISC), oppure da una specifica circuiteria (come nel caso dell'architettura RISC).

Livello 2 - ISA È costituito dall'*Instruction Set Architecture* ed è quindi governato dal linguaggio macchina. Offre visione ed accesso diretto a tutte le risorse fisiche del sistema, tramite una specifica interfaccia di livello. Fornisce un'interfaccia tipicamente *software* (e quindi programmabile) ai livelli superiori. Si può agire al livello 1 tramite interpretazione (microprogramma) o disporre di un'esecuzione diretta a livello 0.

Livello 3 - Sistema Operativo È un'estensione del livello ISA ottenuta aggiungendo servizi che vengono eseguiti (interpretati) da un programma del livello ISA chiamato appunto *Sistema Operativo*. Esso garantisce l'operatività di base del calcolatore, coordinando e gestendo le risorse *hardware* di processamento e di memorizzazione, le periferiche, le risorse e le attività *software* legate ai vari processi in uso, funge da interfaccia con l'utente, consentendo l'impiego di altri *software* d'utente, come le varie applicazioni o le librerie.

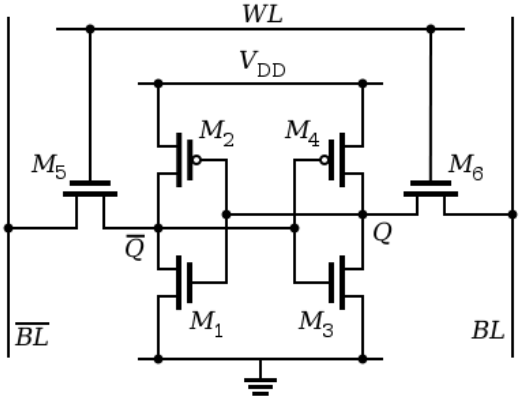
Livello 4 - Assembler A questo livello si fornisce una rappresentazione simbolica di uno dei livelli sottostanti, impiegando sequenze alfanumerice pensate per essere mnemoniche e comprensibili. Infatti i linguaggi binari dei livelli più bassi sono difficili (o praticamente impossibili) da usare per un programmatore. Il programma che traduce da programmi in linguaggio assemblativo a programmi a livello ISA è detto *assembler* o *assemblatore*. A ogni istruzione del linguaggio assemblatore corrisponde un'istruzione del linguaggio macchina che viene eseguita direttamente. I programmi a questo livello non sono usati dal programmatore medio, che deve realizzare programmi applicativi, ma solo dai programmatori di sistema.

Livello 5 - Linguaggi applicativi È caratterizzato dall'impiego di inguaggi come C, C++, *Java, BASIC, LISP, Prolog*, ..., chiamati per l'appunto *linguaggi di alto livello*. Sono impiegati per la realizzazione di programmi applicativi. Il più delle volte la traduzione è affidata a un compilatore, mentre in alcuni casi si usa un interprete.

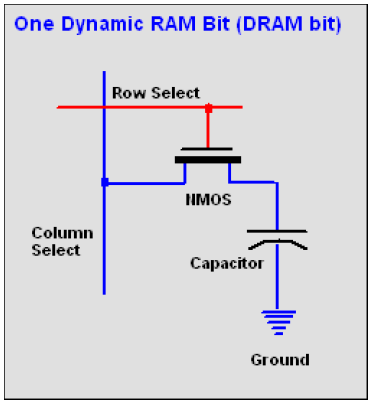
7.1.3 La gerarchia delle memorie

L'architettura di von Neumann si caratterizza per il caricamento del programma in esecuzione nella memoria principale del *computer*, la memoria RAM. L'esecuzione delle istruzioni che si susseguono viene attuata secondo il ciclo *fetch-decode-execute* illustrato nella figura 7.7. Dalla stessa figura appare che ci sono alcuni dati, quelli memorizzati nei registri, che vengono elaborati direttamente a livello di UC, e poi ce ne sono altri che derivano dalla lettura della memoria RAM. Inoltre il programma caricato sulla RAM deve essere reperito dalla *memoria di massa* visibile in figura 7.2, che costituisce l'ambiente all'interno del quale vengono *installati* tutti i programmi in uso all'utente. Come vedremo tra poco l'esistenza di alcuni vincoli di carattere tecnologico e funzionale impongono a queste tre tipologie di memoria (registri, RAM e memoria di massa) delle caratteristiche che sono in qualche misura complementari. In prossimità dell'UC, lavorando a livelli di registri, è necessario disporre di memorie ad accesso veloce, che saranno per forza di cose anche costose. Per contro non è richiesta una elevata capacità di memoria. Man mano che ci si allontana dal cuore del sistema, cioè dalla CPU, è sempre meno importante avere velocità elevate; bisogna però disporre di memorie molto capienti e, di conseguenza, anche molto economiche. Tutto ciò porta a una struttura gerarchica delle memorie, che ora ci prestiamo ad analizzare.

I registri - Abbiamo sottolineato il fatto che i registri sono il primo livello di memoria direttamente impiegato dall'*Unità Centrale* della CPU, al punto che si trovano circuitalmente inseriti all'interno del *case* del microprocessore; nei computer più recenti hanno una dimensione di 32 o 64 *bit*, ed è a questo valore che ci si riferisce per specificare la corrispondente architettura. Poiché costituiscono la memoria immediatamente disponibile per le istruzioni a livello di linguaggio macchina, essi devono operare alle velocità imposte dal ciclo macchina *fetch-decode-execute* e quindi devono essere memorie estremamente veloci. I registri sono realizzati con una tecnologia elettronica denominata *RAM statica* (SRAM), basata sull'impiego dei MOS-FET; questi sono collegati come appare nel circuito di figura 7.10a, che è in grado di memorizzare un singolo bit. Il principio di funzionamento è quello già visto per il circuito *Flip-Flop* incontrato in figura 2.26. Ricordiamo che RAM significa *Random Access*



(a) Memoria RAM statica (SRAM) in grado di memorizzare un bit



(b) Memoria RAM dinamica (DRAM)

Figura 7.10: Circuiti per la realizzazione delle memorie RAM statiche e dinamiche

Memory, ed evidenzia un aspetto tecnologicamente rilevante della memoria, vale a dire la possibilità di accedere a una cella qualunque della stessa (da cui la specificazione *Random*) in un tempo costante. Questa caratteristica non vale, per esempio, per la tecnologia delle memorie a disco, che costituiscono le memorie di massa del tipo *Hard Disk* (HD) di molti *computer*.
Le unità di memoria SRAM sono di tipo *volatile* (si perde la memoria quando il circuito non è più alimentato) e necessita molti componenti (solitamente 6 MOS-FET per bit); è di conseguenza una memoria estremamente costosa e caratterizzata da una bassa densità di memorizzazione (in genere basterebbe un singolo transistor per bit, contro i 6 usati in questa caso). In cambio la tecnologia dei MOS-FET garantisce un bassissimo consumo di potenza e un tempo di accesso che è il migliore realizzabile con le odierne tecnologie; il valore si attesta di $1 - 2$ ns, ma

può arrivare anche a $0,5$ ns; come conseguenza si ottiene un'altissima velocità di trasferimento dati (tipicamente > 12000 MB/s).

La RAM - La struttura schematica del calcolatore riportata in figura 7.2 evidenzia la stretta connessione tra CPU e *memoria principale* di tipo RAM. Su questa memoria, attualmente di tipo volatile, viene infatti caricato il programma in esecuzione secondo il paradigma dell'architettura di von Neumann. Anche la memoria principale deve possedere delle buone caratteristiche di velocità di accesso, ma il costo delle memorie SRAM rende proibitivo il loro impiego come memoria principale. Si ricorre pertanto a una tipologia più economica di memoria RAM, denominata DRAM (*Dynamic RAM*), che è almeno $5 - 10$ volte più lenta ($5 - 10$ ns come tempo di accesso), ma ben 100 volte più economica della SRAM. Poiché sulla memoria principale devono trovare posto, oltre ai dati intermedi della computazione, anche tutti i programmi che sono contemporaneamente in esecuzione, essa deve possedere una capacità rilevante; attualmente, per un *computer* di tipo portatile, la memoria principale è dell'ordine di qualche unità di GB ($2 - 16$ GB).

La *cache* - Una quantità di memoria così elevata non è integrabile all'interno del *chip* del processore, ed è quindi necessario tenere la memoria principale fisicamente all'esterno dello stesso; ciò comporta un ulteriore rallentamento nelle operazioni di lettura e scrittura in memoria principale, che diventa di fatto accessibile in tempi che si attestano nell'intervallo $30 - 90$ ns. A questo punto abbiamo una comunicazione molto veloce ed efficiente tra UC e registri, che consente una rapida esecuzione delle istruzioni, ma una comunicazione $30 - 90$ volte più lenta tra UC e memoria RAM esterna, il che ritarda la lettura delle istruzioni che devono essere eseguite. Questo fatto suggerisce l'introduzione di un ulteriore *livello di memoria* all'interno del *chip* del processore, denominato *cache*, che serve per memorizzare in anticipo le informazioni che l'UC è in procinto di chiedere alla RAM per la successiva esecuzione. Per chiarire il concetto facciamo un esempio.
Supponiamo che l'UC sia in grado di eseguire un'istruzione in 10 ns e di leggere dalla memoria RAM in 60 ns. Immaginiamo che il programma in esecuzione abbia 100 istruzioni, $I(1), I(2), I(3), \ldots, I(100)$. Per leggere la prima istruzione, $I(1)$, l'UC ci mette 60 ns; poi la manda in esecuzione, impiegando 10 ns, e contemporaneamente inizia la lettura di $I(2)$, che dura sempre 60 ns. Ciò significa che il processore, dopo aver eseguito $I(1)$, deve attendere 50 ns prima di poter iniziare l'esecuzione di $I(2)$. In altre parole la cadenza con la quale vengono eseguite le istruzioni non è quella veloce alla portata dell'UC, di 1 istruzione ogni 10 ns, ma quella più lenta di 1 istruzione ogni 60 ns, a causa del rallentamento imposto dalla lettura della RAM esterna. Se introduciamo una memoria *cache* di prestazioni (e capacità) intermedie tra registri e memoria principale esterna, p.es. basata su una tecnologia SRAM che risponde con una velocità di 20 ns, possiamo memorizzare un primo blocco di istruzioni nella *cache* (per esempio $I(1), I(2), \ldots, I(10)$) ed eseguirle subito dopo al tasso di 1 istruzione ogni 20 ns, incrementando la velocià di un fattore pari a 3. La memoria *cache* è dunque una memoria tampone, che serve come contenitore intermedio e provvisorio della RAM nella sua interazione con l'UC.
Per questo motivo a partire dagli anni $'70$ iniziò a diffondersi l'uso della memoria *cache* per velocizzare la lettura dalla memoria principale; in un primo periodo essa era unica, ma nel seguito vennero costituiti più *livelli*, tipicamente due per processori singoli e tre per processori *multicore*, cioè con più di un'UC all'intreno di un unico *chip*. Il primo di questi livelli, L1, sta immediatamente a ridosso dell'UC e ha una capacità relativamente modesta, ma è caratterizzato da una memoria molto veloce e costosa. La *cache* di secondo livello, la L2, è più capiente, più lenta e meno costosa, mentre il terzo livello L3 si usa solitamente nelle realizzazioni *multicore*, ed è una *cache* condivisa tra i vari *core*; la figura 7.11 mostra una fotografia della locazione fisica delle memorie *cache* all'interno di un processore *quad-core*.
La struttura di memoria che si sta delineando per il processore è allora quella di una *gerarchia delle memorie* la quale, partendo dai registri che sono i più veloci e costosi, passa per i tre livelli della *cache*, la RAM e infine la memoria di massa, cui corrisponde la capacità più elevata, ma una velocità di accesso più bassa; quest'ultima è disponibile tanto nella versione elettromeccanica (*Hard Disk*) che in quella a stato solido (*Solid State Disk*). La figura 7.12 offre una visione schematica di questa gerarchia, con un'indicazione di massima delle capacità in gioco. La memoria RAM (fig.7.13a) è solitamente alloggiata nelle immediate vicinanze del processore, su circuiti stampati connessi a pettine sulla scheda madre (fig.7.13b).

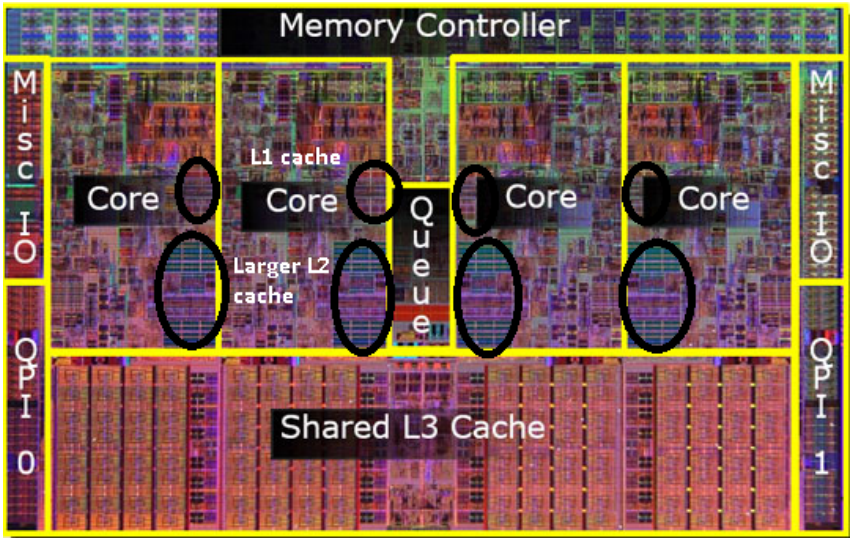


Figura 7.11: Locazione fisica delle memorie *cache* L1, L2 e L3 all'interno di un processore *quad-core*

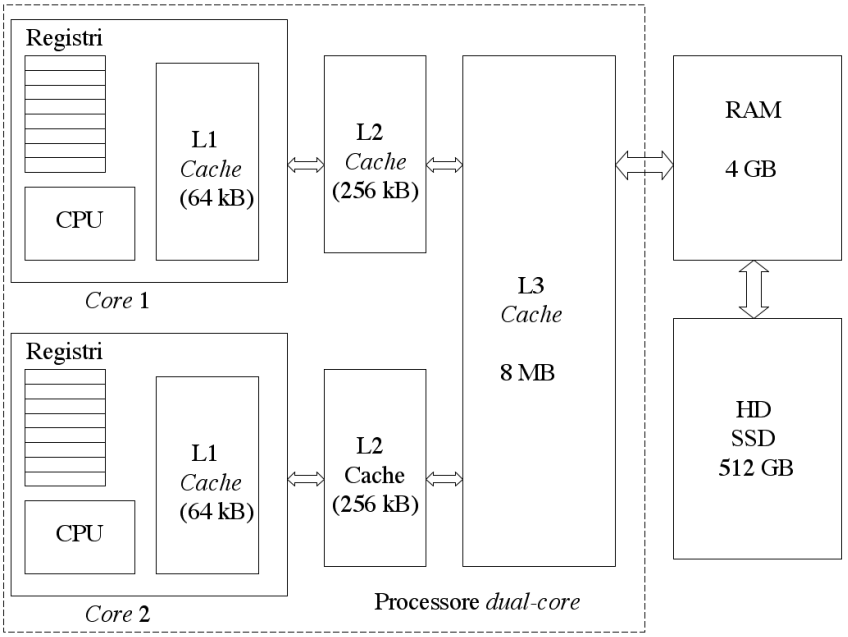
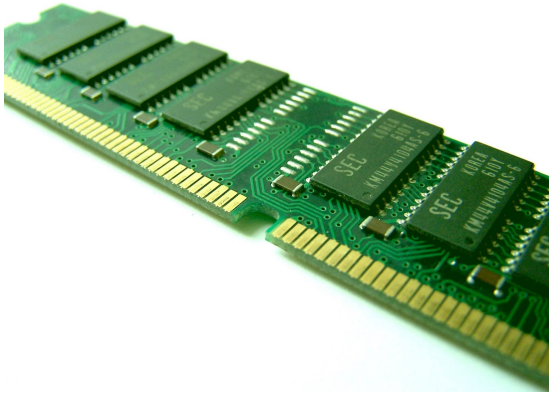
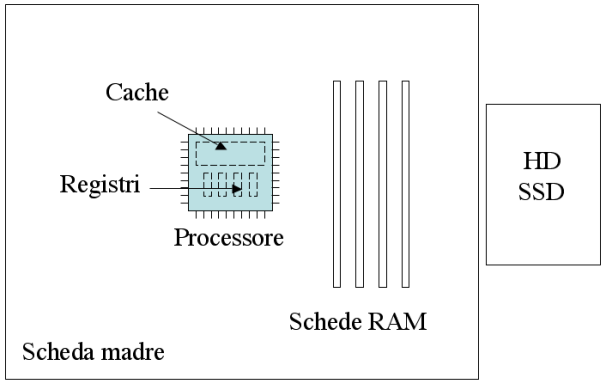


Figura 7.12: Struttura delle memorie *cache* di primo, secondo e terzo livello in un processore *dual-core*; in evidenza anche la memoria principale (RAM) e la memoria di massa (HD-SSD)

La memoria secondaria o di massa - L'ultimo livello della gerarchia di memoria che risiede all'interno della scheda madre è la *memoria di massa*. Su di essa vanno caricati tutti i programmi che vengono installati sul *computer* e che sono quindi a disposizione dell'utente. Ciò implica la necessità di disporre di una memoria di tipo *permanente*, che non perda le informazioni memorizzate quando il circuito non è alimentato.
Alla fine degli anni '50 le memorie di massa erano costruite impiegando i nuclei di ferrite visti in figura 2.24c, che sfruttano la permanenza di una polarizzazione magnetica del nucleo a seguito di una una corrente che scorre in un conduttore che attraversa il nucleo stesso. Successivamente vennero realizzati i primi *dischi rigidi*, o *Hard Disk*, che sono dei dischi di plastica rigida sui quali viene depositato uno stato sottilissimo di una sostanza ferro-

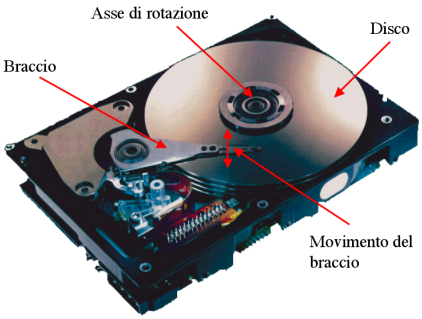


(a) Scheda di memoria RAM; ogni circuito integrato contiene 512 kB

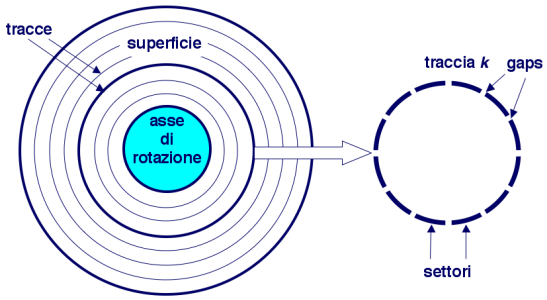


(b) Collocazione delle schede di memoria RAM e della memoria di massa (HD-SSD) sulla scheda madre di un calcolatore

Figura 7.13:



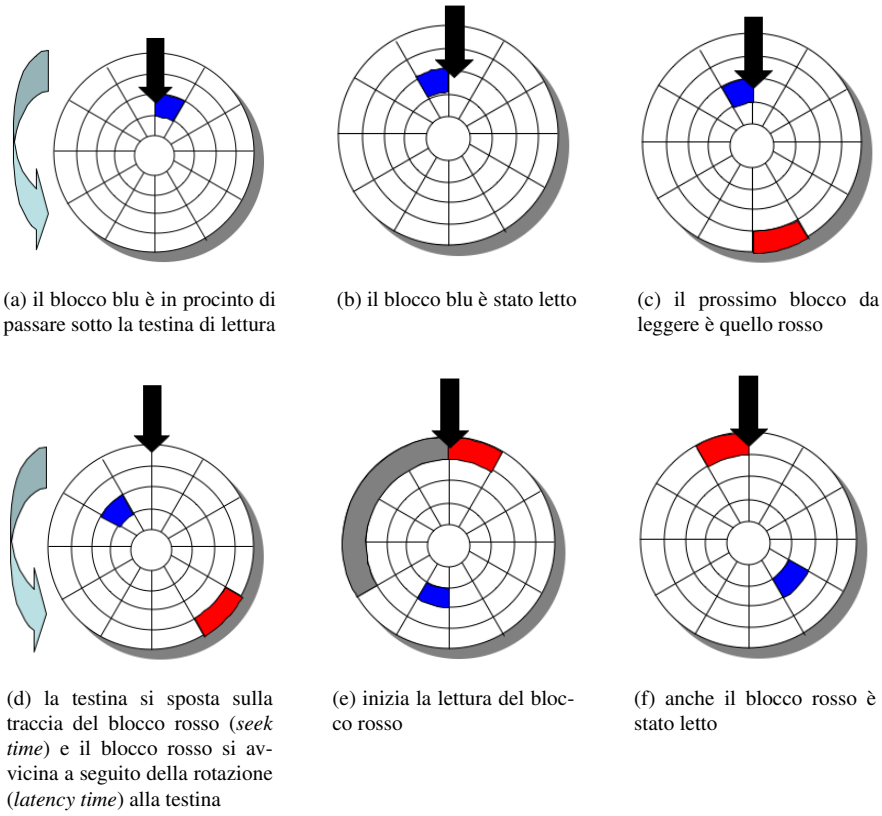
(a) Struttura fisica di un *Hard Disk*



(b) Suddivisione del disco in tracce e settori

Figura 7.14: Struttura interna di un *Hard Disk*

magnetica; essi vengono polarizzati secondo la tecnica illustrata nella figura 2.27a e vengono posti in rotazione a una velocità molto elevata, che varia tipicamente tra i 7000 e i 10000 giri/min. Gli *Hard Disk* sono dispositivi *elettromeccanici*, in quanto la lettura e la scrittura delle informazioni avviene mediante una testina magnetica montata su un braccio oscillante e comandata da servomeccanismi elettronici. La struttura di un *Hard Disk* è illustrata in figura 7.14a; in essa è visibile, oltre al disco e il suo asse di rotazione, anche il braccio che sorregge la testina magnetica di lettura/scrittura. Al disco viene poi attribuito un *formato* magnetico, basato su un certo numero di *tracce* concentriche e *settori*. Una caratteristica importante degli *Hard Disk* è che essi non hanno un indirizzamento sul singolo *byte*, bensì su un blocco la cui grandezza è compresa tra i 512 Byte, che sono un valore classico e standard, e i 4 kB delle ultime realizzazioni. L'accesso al singolo blocco non è effettuato in tempo costante, come nel caso delle memorie RAM, ma in un tempo che dipende dalla posizione della traccia cui il blocco appartiene e del blocco all'interno della traccia. Il tempo complessivo di accesso al disco dipende dal contributo di tre componenti, il *seek time* ($3 - 5$ ms), necessario alla testina per raggiungere la traccia sulla quale si trova il blocco che deve essere letto/scritto, il *latency time* (< 3 ms), necessario per attendere che il blocco passi per effetto della rotazione del disco sotto la testina magnetica e il *transfer time* ($0,006$ ms), necessario per effettuare la lettura/scrittura. Il valore complessivo della lettura varia nell'intervallo $3 - 6$ ms, che è un tempo enorme rispetto ai tempi relativi alle memorie elettroniche; il rapporto è dell'ordine di 10^6 con i registri e di 10^5 con la memoria RAM. Come conseguenza anche per gli *Hard Disk* è necessario prevedere dei livelli di *cache* sulla RAM (*page cache*) o direttamente sull'elettronica del disco rigido (*disk buffer*) che agevolino la lettura dei dati.
Il meccanismo di lettura è illustrato in figura 7.15, tenendo conto del fatto che il disco sta girando in modo antiorario:



(a) il blocco blu è in procinto di passare sotto la testina di lettura

(b) il blocco blu è stato letto

(c) il prossimo blocco da leggere è quello rosso

(d) la testina si sposta sulla traccia del blocco rosso (*seek time*) e il blocco rosso si avvicina a seguito della rotazione (*latency time*) alla testina

(e) inizia la lettura del blocco rosso

(f) anche il blocco rosso è stato letto

Figura 7.15: *seek time* e *latency time* nel funzionamento di un disco rigido

Fig. 7.15a il blocco blu è in procinto di passare sotto la testina di lettura

Fig. 7.15b il blocco blu è stato letto

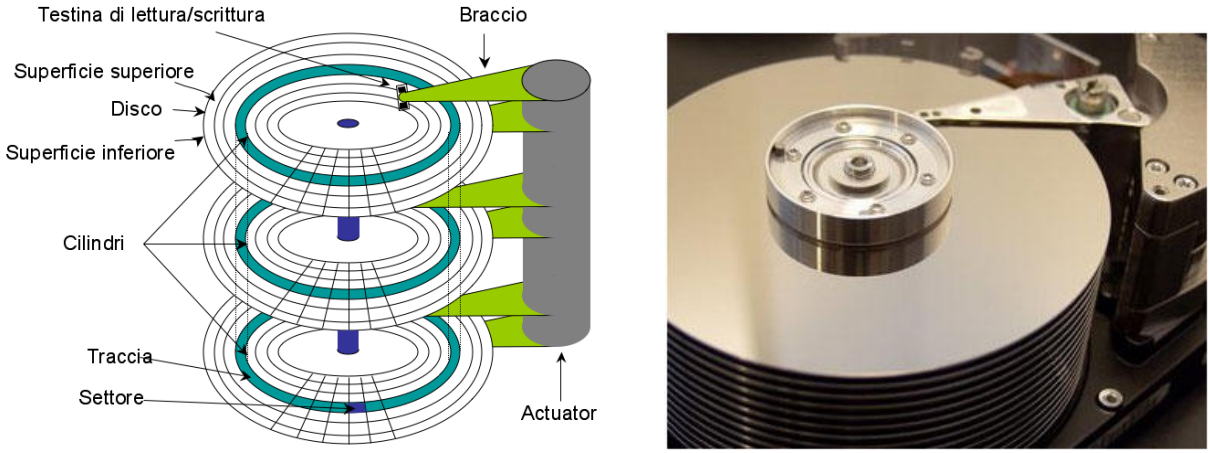
Fig. 7.15c il prossimo blocco da leggere è quello rosso

Fig. 7.15d la testina si sposta sulla traccia del blocco rosso (*seek time*) e il blocco rosso si avvicina a seguito della rotazione (*latency time*) alla testina

Fig. 7.15e inizia la lettura

Fig. 7.15f anche il blocco rosso è stato letto

Per quanto riguarda le altre caratteristiche dei dischi osserviamo che gli *Hard Disk* devono poter contenere tutti i programma in uso all'utente, oltre al *sistema operativo*, che è il programma che gestisce il funzionamento complessivo del *computer*; di conseguenza la capacità di memorizzazione deve essere notevole; nei modelli attuali arrivano a 512 GB per *computer* portatili, e a molti Tera Byte per le macchine più potenti. Nel caso di grossi *server* si usano più dischi rigidi organizzati in banchi, che operano in parallelo (spesso come dischi di *back-up*) ruotando sullo stesso asse (vedi fig7.16). Gli *hard-disk* sono caratterizzati da costi per unità di memoria molto contenuti e sempre in diminuzione.

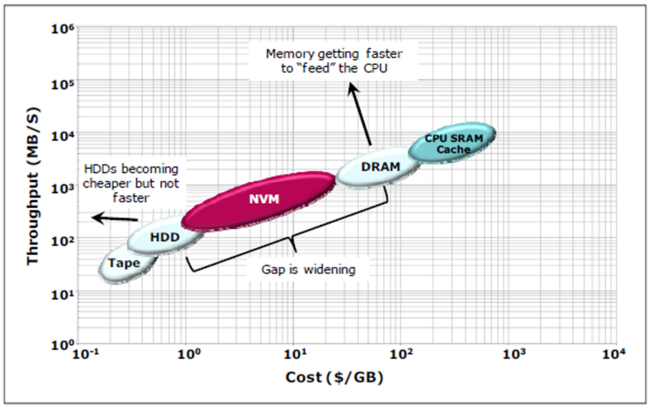


(a) Principio di funzionamento

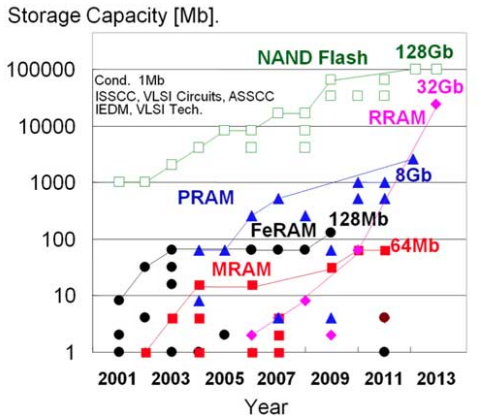
(b) Banco di *herd disk* di un grosso *server*

Figura 7.16: Banco di dischi rigidi che ruotano sullo stesso asse

Se mettiamo in relazione il legame che esiste tra tipologia di memoria, la velocità di gestione dei dati e il costo per unità di GB (espresso in dollari USA), otteniamo il diagramma di figura 7.17a. Al vertice delle prestazioni (e dei costi) ci sono le memorie SRAM, impiegate per i registri e per la *cache* di primo livello. Segue la DRAM della memoria primaria e molto distanziati i dischi rigidi (HDD) e le memorie a *nastro magnetico*, di cui non abbiamo parlato, ma che costituiscono l'ultimo gradino della gerarchia di memoria; essi sono disponibili solo come memoria di massa di *back-up* all'esterno del *computer* e sono caratterizzati da un costo bassissimo.
Come si vede bene dal diagramma, lo spazio tra DRAM e HDD è stato riempito di recente da una nuova tecnologia di memoria, denominata *Non Volatile Memory* (NVM). Si tratta di memorie elettroniche di tipo non volatile, nelle quali si riesce a mantenere l'informazione anche quando il circuito non viene alimentato. Ci sono diverse tecnologie a supporto delle memorie NVM, ma la più matura usata a tutt'oggi è quella delle memorie *flash*, usata in passato solo per apparecchiature sofisticate e ad alto costo (fotocamere digitali professionali), ma largamente diffusa ora per la produzione delle memorie a pennetta USB. Il principio fisico di funzionamento di una memoria *flash* è piuttosto complesso, e possiamo limitarci a dire che sfrutta la possibilità di costruire dei MOS-FET di tipo speciale, nei quali alcune cariche elettriche vengono imprigionate da un secondo *gate* isolato e a potenziale



(a) Velocità di lettura dei dati delle varie memorie della gerarchia rapportata al costo



(b) Capacità raggiunte da varie tipologie di memorie NVM negli ultimi anni

Figura 7.17:

flottante. Recentemente sono state sviluppate anche delle NVM basate sulla sostituzione del dielettrico isolante dei MOS-FET con del materiale ferroelettrico (*Ferroelectric RAM* - FeRAM, F-RAM o FRAM) o addirittura sul fenomeno del *Magnetic Tunnel Junction* (MTJ), fenomeno descrivibile solo a livello di fisica quantistica, che porta alle promettenti memorie *Magnetoresistive* RAM (MRAM). Altri tipi di memoria NVM basati invece sulla variazione controllata di resistenza sono la *Resistive RAM* (RRAM), che sfruttano la proprietà di alcuni ossidi di cambiare il valore della resistenza in modo controllato elettricamente e la *Phase-change* RAM (PRAM), la cui variazione di resistenza deriva da una variazione dello stato cristallino dell'elemento costituente.
La disponibilità delle memorie NVM di tipo *flash*, a costi in veloce diminuzione, ha comportato la progressiva sostituzione dei dischi rigidi elettromeccanici con banchi di memoria elettronici denominati SSD (*Solid State Disk*), almeno per le tipologie di *computer* portatili. I tempi di accesso sono circa 200 volte migliori dei tempi per gli *Hard Disk*, poiché si attestano sui 25 μs; la figura 7.18 ci mostra un confronto tra un HD tradizionale e un suo equivalente nella versione SSD a stato solido.
Anche se la tecnologia elettromeccanica dei dischi rigidi è oramai sorpassata, questo tipo di disco continua a riscuotere un lusinghiero successo a causa del costo in continua diminuzione e alle doti di affidabilità e di maturità raggiunta da questa tecnologia; è comunque forza maggiore la sua progressiva alienazione man mano che saranno disponibili le memorie permanenti a stato solido a prezzi più convenienti.

Figura 7.18: Confroto tra un disco rigido elettromeccanico e la corrispondente versione SSD a stato solido, basata sulle memorie NVM

Le memorie esterne - Facciamo solo qualche brevissimo cenno sulle memorie che si possono connettere esternamente al calcolatore tramite le porte d'ingresso o i dispositivi di lettura CD/DVD, poiché ne abbiamo già parlato precedentemente (vedi figura 2.27b). Per quanto riguarda i DVD possiamo dire che la tecnologia laser è nella fase finale del suo sviluppo e verrà presto soppiantata completamente da supporti basati su memorie NVM. Il motivo del declino è che anche in questo caso la presenza di una tecnologia elettromeccanica per la lettura dei dati, che non ha più altra giustificazione se non che quella economica, visto il bassissimo costo del supporto. Anche per quanto attiene i dischi rigidi esterni, che si usano come unità di *back-up* la loro sostituzione con memorie NVM è solo una questione di tempo. Dal diagramma 7.17a e dalla tendenza delle recenti tecnologie NVM si può intuire che si sta andando verso un'architettura del *computer* in cui la differenza tra memoria principale e memoria di massa del tipo SSD si farà sempre più sfumata, man mano che aumenteranno le prestazioni della tecnologia a stato solido del tipo NVM. È verosimile che tra qualche tempo i dischi rigidi non saranno più usati neanche sui grossi *server*, realizzando anche per questi dispositivi quella sorta di fusione tra i vari strati della gerarchia di memoria, che si distribuirà in modo molto più uniforme tra interno ed esterno della CPU, senza soluzione di continuità e basata tutta su tecnologie elettroniche.

Tipo di memoria	Tempo di accesso	Velocità trasferimento dati MB/s	Costo Euro/GB	Capacità
Registri	$0,5$ ns	$3000 \div 17000$	$110 \div 540$	2 kB
Cache L1	2 ns			64 kB
Cache L2	5 ns			256 kB
Cache L3	$20 \div 50$ ns			8 MB
Memoria principale (RAM)	$30 \div 90$ ns	$1000 \div 6400$	10	4 GB
Non Volatile Memory (MRAM)	35 ns	400	5000	4 MB
Solid State Disk (SSD)	$25 \div 100$ μs	250	$0,7$	512 GB
Hard Disk (HD)	$5 \div 20$ ms	80	$0,1$	512 GB
Dischi laser (DVD)	$0,1 \div 5$ s	$4,5 \div 20$	$0,15$	25 GB
Nastri magnetici	> 10 s	$80 \div 240$	$0,15 \div 0,40$	1 TB

Figura 7.19: Principali caratteristiche delle memorie di un *computer*. Nella parte alta troviamo le memorie più vicine all'UC, più veloci e più costose, e nella parte bassa le memorie più distanti, lente ed economiche

Nella tabella di figura 7.19 riportiamo un quadro riassuntivo delle principali caratteristiche delle memorie di un *computer*, organizzate in una gerarchia che vede nella parte alta le memorie più vicine all'UC, più veloci e più costose, e nella parte bassa le memorie più distanti, lente ed economiche. Si noti il prezzo delle memorie MRAM, che riflette una tecnologia che rimane ancora d'avanguardia.
Concludiamo con la figura 7.20, che offre uno sguardo d'insieme della topologia del calcolatore, che mette in evidenza i dispositivi principali e la loro connessione realizzata tramite *bus*.

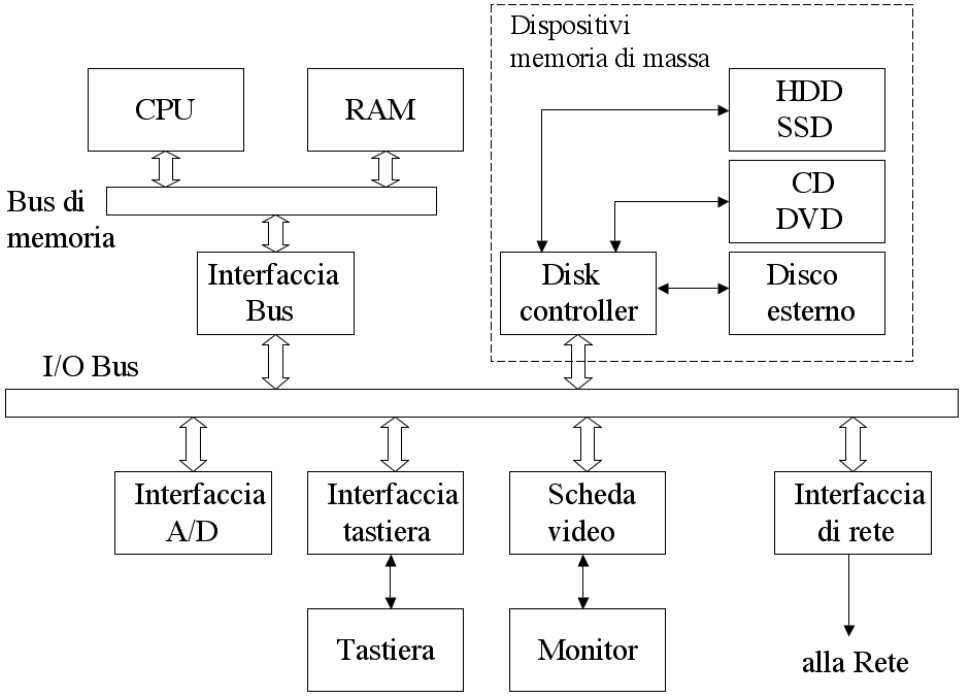


Figura 7.20: Dispositivi principali di un calcolatore

Capitolo 8

Un modello di computazione per il calcolatore

Quando si studia la realtà fisica che ci circonda, seguendo l'approccio scientifico Galileiano, si tenta di condensare all'interno di un *modello* quelle che sono le evidenze sperimentali del particolare sottosistema che si sta analizzando. Un modello è una rappresentazione astratta del sistema, che ne riproduce le caratteristiche, le peculiarità e i comportamenti fondamentali. La definizione del modello va fatta solitamente ammettendo alcune ipotesi generali sul sistema e deducendo le leggi matematiche che sono alla base della descrizione del suo comportamento. Quando si dispone di un modello è possibile fare delle *previsioni* sul comportamento del sistema, sfruttando le equazioni che lo rappresentano.
Per esempio il *modello del corpo rigido*, impiegato nella *meccanica classica*, consente di descrivere il comportamento di un corpo caratterizzato dall'ipotesi che non sia deformabile. Ciò significa che, scelti due punti qualunque x_1, y_1, z_1 e x_2, y_2, z_2 appartenenti al corpo e detta d_{12} la loro distanza iniziale, il vincolo di rigidità è analiticamente espresso dalla relazione:

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 - d_{12} = 0$$

che deve valere per ogni istante successivo a quello iniziale. Del corpo rigido possiamo studiare la sua *cinematica* e la sua *dinamica*, deducendo delle equazioni che ci consentono di fare delle previsioni sul comportamento del sistema. Per esempio è possibile stabilire il punto esatto in cui una sfera indeformabile di ferro, di diametro di 1 cm, cadrà quando lanciata verso l'alto con un'inclinazione di 32° rispetto al piano orizzontale e con una velocità iniziale di $3,5$ m/s.
Qualunque modello offre tuttavia una rappresentazione solo parziale della realtà, poiché tende a evidenziare le sole grandezze fisiche che ci interessano al livello di analisi in cui stiamo operando; esso è legato, in modo indissolubile, alle ipotesi che facciamo sul sistema. Se le ipotesi diventano meno vincolanti, il modello deve essere "esteso" per poter tener conto del nuovo stato di cose. Se per esempio prendiamo la nostra sferetta di ferro e ipotizziamo che si sposti a velocità molto elevate, prossime alla velocità della luce, allora il modello delle meccanica classica non è più sufficiente a descrivere il sistema poiché, per esempio, il diametro della sfera subirà una contrazione nella direzione del moto descritta dalla famosa legge di *Lorenz-FitzGerald*

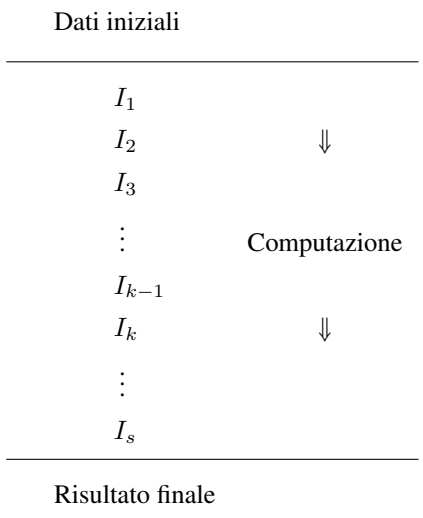
$$D_v = D\sqrt{1 - (v/c)^2}$$

dove D_v è il diametro nella direzione dello spostamento a velocità v, c è la velocità della luce e D è il diametro della sfera in quiete. Il modello deve dunque essere esteso per tener conto delle equazioni della *relatività ristretta*, che offrono una descrizione più accurata del sistema. Assumendo il punto di vista della *Relatività ristretta* possiamo allora affermare che il modello della meccanica classica offre una buona rappresentazione della realtà quando le

velocità in gioco sono trascurabili rispetto alla velocità della luce.
Sulla base di quanto detto finora sembra ragionevole poter costruire un *modello* anche per il calcolatore descritto nel capitolo 7, che specifichi con precisione le ipotesi che riguardano il funzionamento del sistema, in modo che si possano fare delle *previsioni* sul suo comportamento; per esempio sarebbe interessante individuare quali problemi si possono effettivamente risolvere col metodo algoritmico-procedurale anticipato nel paragrafo 1.4 e quali no, oppure conoscere i tempi necessari per ottenere una soluzione di un problema risolubile. In questo capitolo ci occuperemo proprio della costruzione di un modello di computazione per il calcolatore. Come vedremo di modelli ce ne sono tanti, ma si potrebbe dimostrare che essi sono tra loro tutti *equivalenti*, nel senso che portano tutti allo stesso insieme di problemi risolubili o, per esprimersi in modo un po' più tecnico, allo stesso insieme di *funzioni computabili*.
Il modello che sceglieremo è il *modello RAM*, che al contrario degli altri ha il pregio di essere basato sulla struttura architetturale dei computer. Grazie al modello RAM saremo in grado di capire la vera natura della computazione, quali sono i problemi effettivamente risolubili da un calcolatore e quali non lo sono e quali sono i tempi necessari alla risoluzione, in funzione della dimensione del problema.

8.1 Il concetto di algoritmo

Un calcolatore è un dispositivo che serve per eseguire *programmi*. La loro esecuzione è finalizzata alla risoluzione di *problemi* del mondo reale anche molto complessi ed eterogenei tra loro. La *codifica simbolica*, cioè la possibilità di associare un qualunque significato a un simbolo, coniugata con la possibilità offerta dal calcolatore di manipolare i simboli in modo logicamente strutturato, consente di trasformare i problemi del mondo reale in *problemi astratti*, di natura logico-simbolica e descrivibili con gli strumenti della matematica. Il problema astratto viene poi risolto in un tempo finito, avvalendosi dell'approccio *procedurale* o *algoritmico* descritto nella sezione 1.4 e che richiamiamo brevemente: si parte da un insieme di informazioni fornite al sistema di elaborazione dall'esterno, i cosiddetti *dati iniziali*, che vengono successivamente elaborati secondo una procedura ordinata a passi, descrivibile in modo preciso ed esauriente come una sequenza *finita* $I_1, I_2, \ldots, I_s$ di *istruzioni elementari*. Il risultato dell'esecuzione dell'istruzione I_k dipende, oltre che dal tipo di istruzione, anche dai dati d'ingresso al passo k, che sono costituiti da eventuali dati esterni e dai dati che derivano dall'esecuzione dell'istruzione I_{k-1} del passo precedente.



Alla procedura in questione viene attribuito il nome di *algoritmo*; la sua esplicitazione rigorosa in un linguaggio comprensibile alla macchina viene chiamata *programma* $\mathcal{P}$, mentre si riserva il termine di *computazione* al processo che consiste nell'esecuzione dell'algoritmo (del programma) a partire dai dati iniziali. Il risultato della

computazione, cioè i dati generati in uscita al termine della stessa e che devono essere prodotti in tempo finito, consente di esprimere in modo codificato la soluzione al problema del mondo reale.

L'approccio procedurale-algoritmico non è l'unico che si possa concepire per risolvere problemi di natura logico-simbolica; esistono infatti anche altri *paradigmi* di computazione, quali ad esempio le *reti neuronali*, gli *algoritmi genetici*, la *computazione DNA* e la *computazione quantistica*; tuttavia il metodo procedurale è l'unico basato su un solido *modello di computazione*, legato ai lavori di Church, Turing, Gödel e Kleene, sui quali si è poi sviluppata tutta la *teoria della computazione*; essa stabilisce, tramite rigorosi teoremi matematici, i limiti intrinseci dell'approcio procedurale-algoritmico, distinguendo tra *problemi risolubili* (o *predicati decidibili*) e *problemi non-risolubili* (o *predicati indecidibili*). La successiva *teoria della complessità computazionale* si occupa invece di distinguere, all'interno dei problemi risolubili, i gradi di complessità degli stessi, legati al numero di passi elementari che bisogna svolgere per ottenere la soluzione del problema. In quest'ambito la distinzione è tra problemi *trattabili* in un tempo accettabile (in *tempo polinomiale*) e problemi di fatto *intrattabili*, (almeno quando le dimensioni del problema è sufficientemente grande), poiché richiederebbero un numero *esponenziale* di passi, portando a dei tempi di attesa per la soluzione assolutamente inaccettabili (p.es. $3,2$ milioni di anni).

Nello schema procedurale si ipotizza che le istruzioni elementari siano *immediatamente eseguibili*; l'idea di base, molto comune anche nella pratica quotidiana, è che per risolvere un problema complesso sia necessario attuare una strategia la cui descrizione venga specificata da un certo numero di *passi elementari*. Se per esempio voglio uscire dall'ufficio per andare a prendere l'autobus, dovrò aprire la porta se questa è chiusa, scegliere se andare a destra o a sinistra per raggiungere l'uscita dell'edificio, percorrere il corridoio sino alla scalinata, scegliere se salire o scendere dalla stessa ecc. Immaginiamo ora che mi trovi in un edificio che non conosco, e che per raggiungere la fermata abbia in mano un foglio con le istruzioni sul percorso; anche se non ho idea di come sia fatto l'edificio è sufficiente che esegua alla lettera l'elenco delle istruzioni per arrivare alla fermata. Le istruzioni elementari (apri la porta, gira a destra, scendi le scale...) devono però essere alla mia portata e immediatamente eseguibili, senza l'ausilio di ulteriori "istruzioni" supplementari.

Se caliamo questo ragionamento nell'ambito dei calcolatori e se teniamo conto che essi sfruttano le tecnologie elettroniche, l'immediata eseguibilità di un'istruzione implica un'attività a livello circuitale realizzata da un *agente di calcolo* $\mathcal{A}$. Le operazioni eseguite da questo dispositivo sono molto semplici, e possono riguardare una gestione dell'informazione a livello di codice ASCII (nel caso si tratti di manipolazione di simboli su tale alfabeto) oppure di blocchi di *byte* che rappresentano dei numeri seconda una delle notazioni usate (complemento a 2, *floating point*,...) o anche una lettura o scrittura dei dati in una memoria ecc. Di conseguenza le capacità di elaborazione di $\mathcal{A}$ sono necessariamente limitate.

Per l'esecuzione dell'istruzione da parte dell'agente di calcolo potrebbe essere necessaria una *memoria* $\mathcal{M}$ di supporto (memoria RAM), per esempio per memorizzare risultati intermedi di una computazione, che può essere arbitrariamente grande.

L'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma $\mathcal{P}$ avviene sulla trama di un *tempo discreto*, che viene specificato da un orologio interno del calcolatore (*clock*). Il tempo discreto corrisponde a una cadenza temporale prefissata e costante (p.es. 1 miliardesimo di secondo) in corrispondenza della quale possono essere effettuate le operazioni elementari a carico delle circuiteria. In altre parole il sistema rimane "congelato" tra due istanti di tempo discreto successivi. Si osservi che la modalità di interazione in tempo discreto è alternativa a una modalità in *tempo continuo*, che era invece prerogativa dei vecchi sistemi analogici, nei quali le varie grandezze variano con continuià nel tempo.

Un'altra ipotesi implicita che si fa nella realizzazione di un calcolatore è che l'interazione tra l'agente di calcolo $\mathcal{A}$ e il programma $\mathcal{P}$ sia *deterministica*. Ciò significa che, a partire dallo stesso insieme di dati iniziali, l'esecuzione di un insieme specificato di istruzioni (programma) porta sempre allo stesso risultato finale. Tale modalità si contrappone a una computazione nella quale esistano dei meccanismi *aleatori*, in cui sia possibile scegliere tra più di un percorso per la computazione a partire dalle stesse condizioni iniziali.

Per quanto riguarda invece la natura delle istruzioni $I_1, I_2, \ldots, I_s$, possiamo osservare che ogni istruzione I_j deve appartenere a un certo insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni elementari che la circuiteria di $\mathcal{A}$ è in grado di svolgere. Nella logica dell'approccio procedurale le informazioni elementari, proprio in quanto tali, devono essere necessariamente "poche". Come abbiamo visto, la struttura architetturale acquisita dai vari sistemi porta a distinguere tra due filosofie, quelle $CISC$, che sta per *Complex Instruction Set Computer*, e quella $RISC$, che significa *Reduced Instruction Set Computer*. Nel primo caso le istruzioni dette elementari consentono di svolgere

operazioni che sono comunque relativamente complesse e articolate, come la lettura di un dato in memoria, la sua modifica e il suo salvataggio direttamente in memoria tramite una singola istruzione. Si è però osservato che questa impostazione risulta poco vantaggiosa, poiché offre in genere costi maggiori e prestazioni modeste, visto che i tempi di decodifica e di esecuzione sono in generale maggiori anche per le istruzioni più semplici. Nel caso dell'architettura $RISC$ c'è invece un insieme molto ridotto di istruzioni effettivamente elementari (lettura e scrittura in memoria, copia di un dato, somma, sottrazione ecc.), e ciò consente di far lavorare i processori in modo più efficiente. A prescindere dal fatto che si usi un insieme $CISC$ o $RISC$, è ovvio che tale insieme debba avere una dimensione finita, anche se non è però necessario specificare una *limitazione superiore* per k.
Un altro elemento importante da considerare tra le ipotesi del calcolo procedurale-algoritmico è relativa alla dimensione dei dati di ingresso; benché non abbia senso porre alcuna limitazione ad essa, è tuttavia doveroso considerarla come una quantità finita; lo stesso può dirsi anche per la lunghezza della computazione: non si può porre un limite al numero di istruzioni che vengono eseguite prima di arrivare alla fine del programma, visto che tale numero potrebbe anche essere (molto) maggiore di n nel caso il programma contempli dei *cicli*. Tuttavia, affinché il sistema funzioni bene per le finalità per le quali esso è stato costruito e segua lo spirito della definizione di algoritmo, sarebbe auspicabile che la computazione durasse un tempo finito, anche se non limitabile. Come vedremo, però, saremo costretti ad accettare nel modello i casi in cui la computazione incappa in un ciclo infinito o *loop*, dal quale non si può uscire. In tal caso la macchina continua a girare all'infinito, senza pervenire mai all'esecuzione di un'istruzione che porti a uno stop; l'utente interpreta questo fatto come un blocco della computazione e deve forzare dall'esterno l'uscita dal programma oppure, nei casi peggiori, riavviare la macchina, perdendo in entrambi i casi tutto il lavoro svolto. Possiamo allora individuare il seguente elenco delle caratteristiche associate alla nozione informale di algoritmo:

Nozione informale di algoritmo

1)	insieme di istruzioni $I_1, I_2, \ldots, I_s$ di lunghezza finita	$\mathcal{P}$ (*programma*)
2)	c'è un *agente di calcolo*	$\mathcal{A}$ (circuiteria)
3)	c'è a disposizione della *memoria*	$\mathcal{M}$ (memoria RAM)
4)	$\mathcal{A}$ interagisce con $\mathcal{P}$ in *modalità discreta*	(macchina discreta)
5)	$\mathcal{A}$ interagisce con $\mathcal{P}$ in *modalità deterministica*	(macchina deterministica)
	Bisogna fissare un limite finito:	
6)	sulla dimensione dei dati d'ingresso?	NO
7)	sulla dimensione dell'insieme $\mathcal{I} = \{I_1, I_2, \ldots l_k\}$ di istruzioni?	NO
8)	sulla dimensione della memoria?	NO
9)	sulla capacità di computazione di A?	SÌ
10)	sulla lunghezza della computazione?	NO
	Tuttavia:	
11)	sono ammesse computazioni con un numero infinito di passi (computazioni non terminanti)	

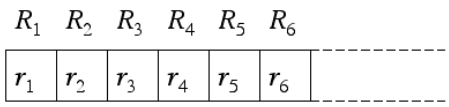
ALGORITMO + 11) = PROGRAMMA

All'algoritmo descritto dai punti $1 - 10$ siamo dunque costretti ad aggiungere l'ipotesi di una computazione in qualche caso non terminante; ciò che ne esce rappresenta la proiezione del concetto astratto di algoritmo, che porta sempre a una soluzione in un tempo finito, sulla realtà del modello effettivo di computazione, che in certe sfortunate situazioni porta a un programma $\mathcal{P}$ che cicla all'infinito. Senza entrare nei dettagli del problema relativo al punto 11) possiamo dire solo che, se lo escludessimo pretendendo di avere a che fare con un modello di calcolo basato sulle sole computazioni terminanti, allora ci si troverebbe in una situazione in cui esisterebbero dei problemi palesemente risolubili mediante approccio procedurale, ma che *non* potrebbero essere risolti all'interno del nostro modello.

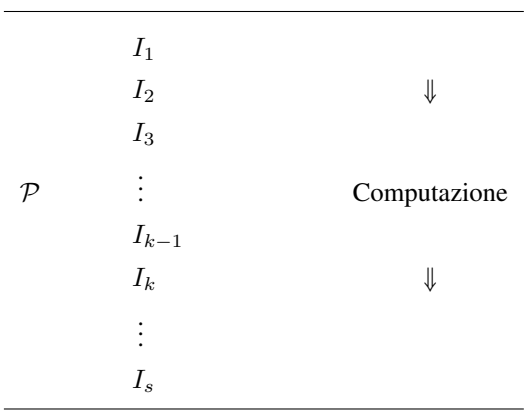
8.2 Il modello RAM

Nel paragrafo 1.3 abbiamo visto che l'informatica nacque sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica, centrati sulla risoluzione del famoso *Entscheidungsproblem* (Problema della decisione). La sua risoluzione, in senso negativo, da parte di *Alonzo Church* e di *Alan Turing* nel 1936, costituisce il punto di partenza della moderna *Teoria della Computabilità*. Tuttavia il lavoro di Church era molto astratto e di difficile comprensione, mentre il modello della *Macchina di Turing* fece ben presto breccia e divenne in breve tempo il modello di riferimento. Ciononostante, questi approcci alla computazione erano stati creati non per rispondere alle esigenze di modellare il funzionamento di un *computer*, che all'epoca non esisteva ancora, ma per dare soluzione a un problema della logica. Ci si trovò dunque, ben presto, nella spiacevole situazione di avere dei *computer* pienamente funzionanti, associati però a dei modelli di computazione che non avevano nulla a che fare con le linee architetturali del sistema che intendevano rappresentare. Oltre ai due modelli citati se ne introdussero successivamente anche altri (Gödel-Kleene, Gödel-Herbrand-Kleene, Post, Markov), tutti però basati su procedimenti di calcolo molto astratti, che non trovavano rispondenza diretta con quanto attuato dai *computer* reali. Solo nel 1963 Shepherdson e Sturgis introdussero un modello, che chiameremo *modello RAM*, che prendeva spunto dalla effettiva struttura del calcolatore moderno, basata come abbiamo visto sull'architettura di Von Neumann e sulla memoria RAM. Passiamo ora alla descrizione di tale modello.

Il modello RAM è basato su un *nastro di memoria di lunghezza infinita*, che rappresenta una idealizzazione delle memorie del calcolatore, e da un *insieme di istruzioni*, che rappresentano le istruzioni in linguaggio macchina (si veda la figura 8.1a). La memoria è costituita da celle che contengono dei numeri naturali r_i, e dunque $r_i \in \mathbb{N}$. In questo senso c'è una differenza concettuale con i computer reali, che come noto lavorano sulla base di un alfabeto binario; tuttavia tale differenza non è rilevante, poiché una qualunque stringa binaria può essere interpretata come numero intero e qualunque numero intero può avere una rappresentazione binaria. L'aspetto concettualmente rilevante è, semmai, il fatto che ogni cella, contenendo un numero intero non limitabile superiormente, è associata a una quantità d'informazione a sua volta non limitabile.



(a) Il nastro di memoria del modello RAM



(b) Il programma $\mathcal{P}$ associato al nastro

Figura 8.1: Gli elementi del modello RAM

Il contenuto delle celle può essere modificato da delle *istruzioni* previste dal modello, che sono state scelte in modo tale da costituire un insieme in qualche senso minimo. Tali istruzioni sono impiegate per realizzare un *programma* $\mathcal{P} = I_1, I_2, ..., I_s$ (fig. 8.1b), che a partire da una *configurazione iniziale* del nastro porta a una *configurazione finale*, dalla quale si ricava il *risultato* della *computazione*; quest'ultima corrisponde all'esecuzione, passo passo, delle istruzioni del programma.

Analizziamo ora le istruzioni, che sono solamente le seguenti quattro:

Istruzione di azzeramento - $Z(n)$ comporta l'azzeramento del contenuto della cella R_n.
Esempio: Se applico l'istruzione $Z(2)$ al nastro della prima riga di figura 8.2 azzero il contenuto della seconda cella (fig.8.2a)

Istruzione di incremento - $S(n)$ comporta l'incremento del contenuto della cella R_n di un'unità.
Esempio: Se applico l'istruzione $S(5)$ al nastro della seconda riga di figura 8.2 incremento di 1 il contenuto della quinta cella (fig.8.2b).

Istruzione di trasferimento - $T(m,n)$ comporta la copia del contenuto della cella R_m nella cella R_n.
Esempio: Se applico l'istruzione $T(2,4)$ al nastro della terza riga di figura 8.2 copio il contenuto della seconda cella nella quarta (fig.8.2c).

Le prime tre istruzioni si chiamano *aritmetiche*, poiché operano manipolando numeri naturali. Usando solamente questo tipo di istruzioni non sarebbe però possibile risolvere problemi con un minimo di interesse pratico, poiché nella vita reale ci si trova sempre nella situazione di dover scegliere strategie diverse a seconda che alcune condizioni siano o meno soddisfatte. In altre parole è necessario introdurre un'istruzione *logica* che ci consenta di aprire percorsi diversi alla computazione.

Istruzione di salto condizionato - $C(m,n,q)$ L'esecuzione di tale istruzione implica il controllo del contenuto delle celle R_m e R_n; se $r_m = r_n$ l'esecuzione del programma continua con l'istruzione I_q, altrimenti si continua con l'istruzione successiva.
Esempio: Se applico l'istruzione $C(1,6,9)$ al nastro della quarta riga di figura 8.2, poiché le celle 1 e 6 hanno lo stesso contenuto, la computazione continua con l'istruzione I_9 del programma (fig.8.2c).

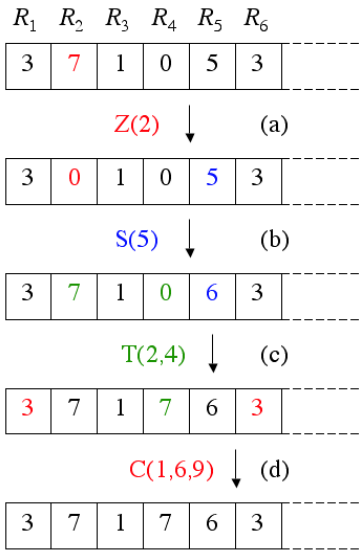


Figura 8.2: Applicazione di alcune istruzioni al nastro di memoria

Vediamo di delineare in modo più preciso i dettagli della computazione in modo che, a partire dal programma e dalla configurazione iniziale, si sappia con precisione quali sono i passi da eseguire e come si ottiene il risultato finale. Lo facciamo evidenziando alcuni punti chiave, che vengono di seguito illustrati.

Computazione - Per eseguire la computazione RAM bisogna fornire un *nastro*, caricato con una *configurazione iniziale* costituita da una sequenza $a_1, a_2, a_3, ...$ di numeri naturali; se $\mathcal{P} = I_1, I_2, ..., I_s$ è il programma

associato alla macchina, la computazione inizia eseguendo l'istruzione I_1. Dopo averla eseguita la macchina RAM dovrà eseguire la *prossima istruzione* (I_{NEXT}) finché si raggiunge uno STOP (se mai si raggiunge).

Prossima istruzione - Se I_k è l'istruzione corrente, la prossima istruzione da eseguire I_{NEXT} è così definita:

- Se I_k è un'istruzione aritmetica $I_{NEXT} = I_{k+1}$ (fig. 8.3a)
- Se $I_k = C(m, n, q)$ si ha $I_{NEXT} = \begin{cases} I_q & \text{se } r_m = r_n \quad \text{(fig. 8.3b)} \\ I_{k+1} & \text{se } r_m \neq r_n \quad \text{(fig. 8.3c)} \end{cases}$

La computazione ha un flusso regolare quando si ha a che fare con istruzioni aritmetiche, oppure quando si incontra un'istruzione di salto condizionato e vale la condizione $r_m \neq r_n$. Se viceversa si ha $r_m = r_n$ la computazione salta all'istruzione I_q.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
$I_k = S(3)$	$I_k = C(3, 5, 9)$	$I_k = C(3, 5, 9)$
$\boldsymbol{I(k+1)}$	.	$\boldsymbol{I(k+1)}$
.	$\boldsymbol{I(9)}$	.
.	.	.
I_s	I_s	I_s

(a) Prossima istruzione nel caso di I_k istruzione aritmetica

(b) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m = r_n$

(c) Prossima istruzione nel caso di $I_k = C(m, n, q)$ con $r_m \neq r_n$

Figura 8.3: I casi possibili per la prossima istruzione (in grassetto)

STOP della computazione - La computazione si ferma per due possibili motivi:

- È stata eseguita $I_k = I_s$ aritmetica, oppure $I_k = I_s = C(m, n, q)$ con $r_m \neq r_n$ (fig. 8.4a e 8.4b)
- È stata eseguita $I_k = C(m, n, q)$ con $r_m = r_n$ e $q > s$ (8.4c)

La prima delle due possibilità corrisponde all'interruzione della computazione per mancanza di altre istruzioni, in quanto è stata eseguita l'ultima istruzione del programma; nel secondo caso, invece, si ha un'uscita forzata poiché si è ottenuto il risultato richiesto e non è più necessario procedere oltre con la computazione. Ciò si ottiene richiedendo l'esecuzione di un'istruzione I_q il cui indice q non figura nell'elenco delle istruzioni del programma; ciò porta (convenzionalmente) a una terminazione della computazione. Nell'esempio di figura (8.4c) è stato posto $q = 99$; ciò fa intendere immediatamente che se vale la condizione $r_m = r_n$, allora ci sarà uno STOP nella computazione, poiché risulta evidente che gli esempi che faremo non avranno mai un numero di istruzioni superiore a 99.
Altro esito possible per la computazione è che essa entri in un ciclo e non giunga mai allo STOP. In questo caso si parla di computazione non terminante o *divergente*.

I_1	I_1	I_1
I_2	I_2	I_2
.	.	.
.	.	$I_k = C(3,5,99)$
.	.	.
$I_s = I_k = S(3)$	$I_s = I_k = C(3,5,9)$	I_{37}
(a) STOP della computazione con I_s=I_k aritmetica	(b) STOP della computazione con I_k=I_s=$C(m,n,q)$ e con $r_m \neq r_n$	(c) STOP della computazione con I_s=I_k=$C(m,n,q)$ e con r_m=r_n e q>s

Figura 8.4: I casi possibili per lo STOP della computazione

Configurazione iniziale - È la configurazione dalla quale si parte per effettuare la computazione. Se $a_1, a_2, ..., a_n$ sono i dati d'ingresso, per convenzione essi vengono posti all'inizio del nastro, lasciando a 0 tutte le altre (infinite) celle di memoria (fig. 8.5).

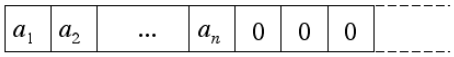


Figura 8.5: Configurazione iniziale del nastro caricato con i dai iniziali $a_1, a_2, ..., a_n$

Configurazione finale - È la configurazione che si ottiene alla fine della computazione. Per convenzione il valore b calcolato dalla computazione è il contenuto della prima cella. Ciò accade ovviamente nel solo caso in cui la computazione termini.

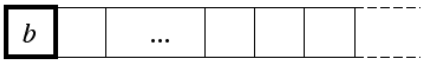
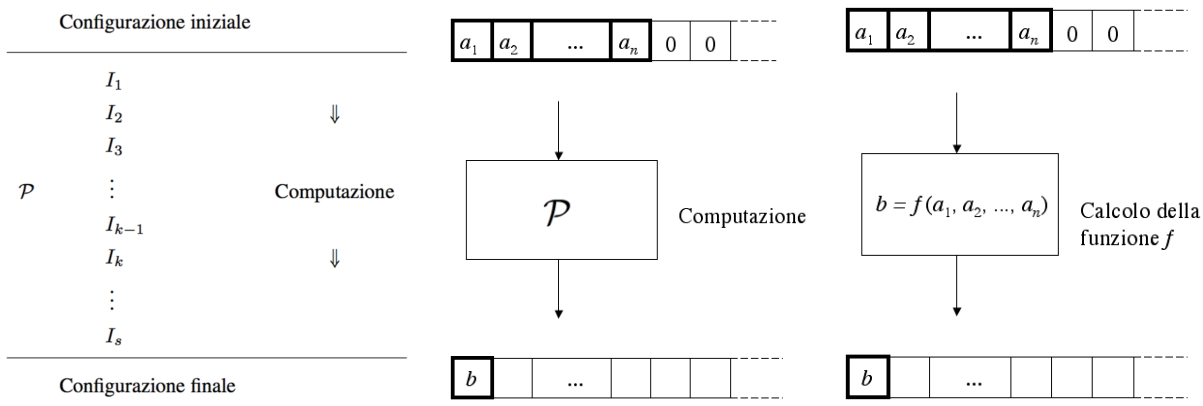


Figura 8.6: Configurazione finale del nastro: il contenuto della prima cella viene considerato il valore calcolato

Convergenza - Indichiamo con la notazione $\mathcal{P}(a_1, a_2, ..., a_n)$ la computazione del programma $\mathcal{P}$ a partire dalla configurazione iniziale $a_1, a_2, ..., a_n$. Se tale computazione termina diciamo che c'è stata *convergenza*, e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\downarrow$. Se b è il contenuto della prima cella alla fine della computazione diciamo che la computazione è andata a convergenza su b e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\downarrow b$. Se la computazione non termina diciamo che si è avuta una *divergenza* e scriviamo $\mathcal{P}(a_1, a_2, ..., a_n)\uparrow$

Analizziamo ora il significato della computazione del programma $\mathcal{P}$=$\{I_1, I_2, ..., I_s\}$ da un punto di vista più astratto. Sappiamo che a partire dalla configurazione iniziale del nastro si perviene alla configurazione finale, che corrisponde al *risultato* della computazione, cioè dell'esecuzione, passo passo, delle istruzioni del programma (fig.8.7a). La computazione è quindi un procedimento astratto il quale, a partire da una certa configurazione iniziale $x_1, x_2, ..., x_n$ restituisce in un tempo finito un certo valore y, leggibile sulla prima cella del nastro (fig.8.7b). Ma un procedimento che associ un numero intero y a un'n-pla $x_1, x_2, ..., x_n$ di interi corrisponde al calcolo della funzione $y = f(x_1, x_2, ..., x_n)$, che è una funzione $f : \mathbb{N}^n \rightarrow \mathbb{N}$ (fig.8.7c).
Invertendo i termini della questione possiamo affermare che, volendo *calcolare* la funzione y=$f(x_1, x_2, ..., x_n)$ possiamo far uso del programma $\mathcal{P}$. Se si vuole calcolare $f(a_1, a_2, ..., a_n)$, mettiamo i valori $a_1, a_2, ..., a_n$ come configurazione iniziale e facciamo partire la computazione; a seguito della convergenza dobbiamo trovare il valore



(a) La computazione del programma $\mathcal{P}$ corrisponde all'esecuzione delle sue istruzioni

(b) La computazione del programma $\mathcal{P}$ dal punto di vista astratto

(c) La computazione del programma $\mathcal{P}$ corrisponde al calcolo di una certa funzione f

Figura 8.7: Il significato della computazione RAM

b nella prima cella. Si osservi che la funzione potrebbe non essere definita per qualche n-pla d'ingresso, per esempio per la $a_1^*, a_2^*, ..., a_n^*$. È ovvio che in questa circostanza non possiamo far convergere la computazione, perché il valore letto nella prima cella alla fine della stessa verrebbe interpretato come *il* valore della funzione, che invece non è definita. L'unica possibilità che rimane è allora quella di far divergere la computazione nel caso in cui la funzione non sia definita.

Calcolo di una funzione tramite un programma Diciamo che il programma $\mathcal{P}$ RAM-calcola la funzione f se, $\forall a_1, a_2, ..., a_n, b$

$$\mathcal{P}(a_1, a_2, ..., a_n) \downarrow b \quad \Leftrightarrow \quad \begin{cases} a_1, a_2, ..., a_n \in \text{Dom}(f) \\ \\ b = f(a_1, a_2, ..., a_n) \end{cases}$$

il che implica che $\mathcal{P}(a_1, a_2, ..., a_n)\uparrow$ se e solo se $a_1, a_2, ..., a_n \notin \text{Dom}(f)$

Funzione calcolabile Una funzione f si dice *calcolabile* se esiste un programma $\mathcal{P}$ che la RAM-calcola.

Potrebbe sembrare una forzatura il fatto che, per esprimere il funzionamento e l'attività di un calcolatore, si debba ricorrere al linguaggio delle funzioni. In realtà ci si rende subito conto che questo è il linguaggio corretto per rappresentare qualunque attività del computer. Esso è infatti una macchina discreta che lavora in tempo discreto; immaginiamo allora di fare una "fotografia" dei suoi circuiti in un certo istante discreto t_0. Poiché il computer è costituito da milioni di transistors, ciascuno dei quali lavora secondo una logica binaria, in linea di principio potremmo fare una lista ordinata di tali transistors, scrivendo "0" o "1" a seconda che, all'istante t_0, il componente sia nello stato di piena conduzione o di interdizione. Il lungo elenco di zeri e uni corrisponde a un vettore binario, con un numero di coordinate uguale al numero di transistors, che descrive lo *stato* della macchina; indichiamo tale vettore con la notazione $a_1, a_2, ..., a_n$. Se passiamo ora all'istante successivo t_1, ci sarà stata un'evoluzione dello stato a seguito delle istruzioni del programma. Supponiamo per esempio che si stia spostando la freccia del *mouse* sullo schermo; potrebbe allora succedere che in t_0 un certo *pixel*, a 256 livelli di colore, fosse p.es. color bianco, caratterizzato dalla codifica 00001011, che corrisponde al numero 11 nella scala $0...255$. Durante lo spostamento, all'istante t_1 il *pixel* diventa color grigio scuro, e viene codificato dal vettore 11011010 che corrisponde al numero 218. Possiamo allora affermare che la descrizione del funzionamento del *pixel* in questione è basata sul calcolo

della funzione $f(a_1, a_2, ..., a_n) = 218$. È ovvio che, secondo questo approccio, serve una funzione per ciascun *pixel*, ma questo è un problema tutto quantitativo, legato alla circostanza che ci sono moltissimi *pixel*. Facciamo ora alcuni esempi di costruzione di programmi elementari che servono per spiegare il funzionamento del modello.

Esempi

Esempio 8.1. Si voglia scrivere un programma che calcola la funzione $f(x) = 0,\ \forall x$.

Si tratta di partire dalla configurazione iniziale $x, 0, 0, \ldots$ per giungere alla configurazione finale $0, \ldots$, qualunque sia il valore di x. Il "programma" è in questo caso costituito dalla sola istruzione $Z(1)$, che porta a zero la prima cella, qualunque sia il contenuto della stessa.

Esempio 8.2. Scriviamo un programma che calcola la funzione $f(x) = 3,\ \forall x$.

Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $3, \ldots$, qualunque sia il valore di x. Il programma è

1	$Z(1)$	(porta a zero la prima cella, qualunque sia x)
2	$S(1)$	(incrementa la prima cella)
3	$S(1)$	(incrementa la prima cella)
4	$S(1)$	(incrementa la prima cella)

Si noti che si può ottenere la stessa funzione incrementando per tre volte il contenuto di una qualunque cella diversa dalla prima (p.es. la seconda) e usando l'istruzione $T(m, n)$ per trasferire il risultato

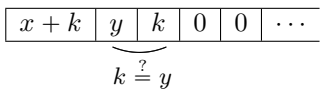
1	$S(2)$	(incrementa la seconda cella)
2	$S(2)$	(incrementa la seconda cella)
3	$S(2)$	(incrementa la seconda cella)
4	$T(2, 1)$	(copia nella prima cella il contenuto della seconda)

Esempio 8.3. Si voglia scrivere un programma che calcola la funzione $f(x) = x + 3,\ \forall x$.
Partendo dalla configurazione $x, 0, 0, \ldots$ si deve giungere alla configurazione finale $x + 3, \ldots$, qualunque sia il valore di x. Il programma è

1	$S(1)$	(incrementa la seconda cella)
2	$S(1)$	(incrementa la seconda cella)
3	$S(1)$	(incrementa la seconda cella)

Esempio 8.4. Si voglia scrivere un programma che calcola la funzione $f(x, y) = x + y,\ \forall (x, y)$.

Partendo dalla configurazione $x, y, 0, \ldots$ si deve giungere alla configurazione finale $x + y, \ldots$, qualunque siano i valori di x e y. Si noti che non possiamo inserire y comandi $S(1)$, poiché non conosciamo a priori il valore di y. Bisogna allora ricorrere alla seguente strategia: si incrementa progressivamente x, in modo che diventi $x + 1$, $x + 2$, $x + 3$, ..., $x + k$,... memorizzando nel contempo il valore corrente di k su una delle celle libere. Quando si giunge al valore k tale che $k = y$, allora nella prima cella c'è $x + y$. Si osservi che il nodo della computazione è il controllo $k \stackrel{?}{=} y$. Se $k \neq y$ dobbiamo continuare a incrementare k; se viceversa $k = y$ allora abbiamo concluso. Lo stato corrente della computaizone è allora

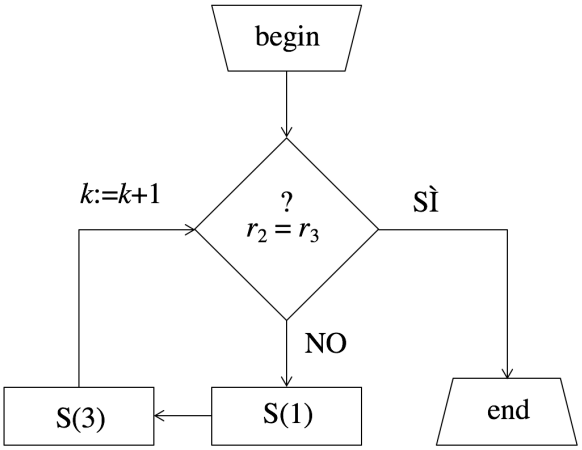


Per poter intraprendere due percorsi diversi di computazione a seconda che k sia o meno uguale a y, è necessario introdurre l'istruzione di salto condizionato $C(m, n, q)$, con m e n indirizzi di memoria dove sono memorizzati k e y. Si noti peraltro che il controllo deve essere svolto immediatamente, poiché potrebbe accadere che sia $y = 0$,

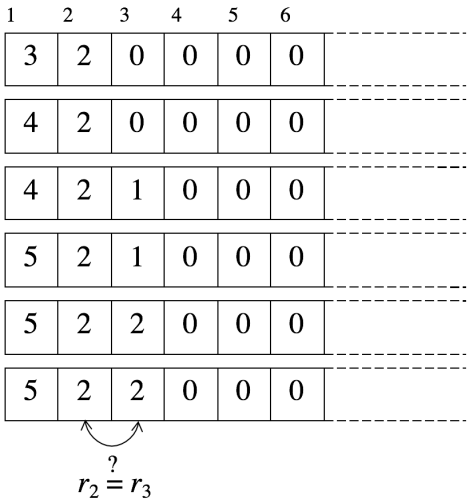
nel qual caso avremmo già terminato. Sarà dunque necessario iniziare con un controllo del tipo $C(2,3,99)$, che ci fa concludere la computazione se il contenuto della seconda cella coincide con quello della terza, cioè se $k = y$. Se viceversa $k \neq y$ si deve incrementare k, cioè $k := k + 1$[1] nella prima cella ($S(1)$) e nella terza cella ($S(3)$). A questo punto la nuova configurazione sarà $x + k + 1, y, k + 1, 0, 0, \ldots$, e dovremo rifare il controllo, per verificare se $k + 1 = y$ o meno. Senza inserire una nuova istruzione $C(2,3,99)$ nel programma, il controllo può essere fatto tornando alla prima istruzione $C(2,3,99)$ mediante un'istruzione di *salto incondizionato* del tipo $C(1,1,1)$; poiché è sempre vero che il contenuto della prima cella è uguale a sè stesso, si riaccede alla prima istruzione e si effettua nuovamente il controllo. In questo modo si crea un *ciclo*, dal quale si esce solo quando è verificata la condizione $r_2 = r_3$.

1	$C(2,3,99)$	(controllo se $k = y$)
2	$S(1)$	($k := k + 1$ nella prima cella)
3	$S(3)$	($k := k + 1$ nella terza cella)
4	$C(1,1,1)$	(salto incondizionato alla prima istruzione)

In figura 8.8a viene riportato il *diagramma di flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso, mentre in figura 8.8b viene riportata la sequenza degli stati di memoria nella somma $3 + 2$.



(a) Diagramma a flusso del programma che calcola la funzione $f(x,y) = x + y$



(b) Successione degli stati per il calcolo della somma 3+2

Figura 8.8:

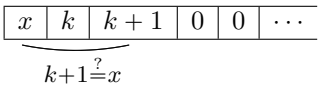
Esempio 8.5. Si voglia scrivere un programma che calcola la funzione $f(x) = x - 1$ sui numeri naturali.

Essa si indica col simbolo $\dot{-}$ e si definisce nel modo seguente

$$f(x) = x \dot{-} 1 = \begin{cases} x - 1 & x > 0 \\ 0 & x = 0 \end{cases} \quad \forall x$$

Per realizzare questa funzione dobbiamo costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Una possibilità è quella di avere una configurazione corrente del tipo

[1] Il simbolo ":=" si legge "diventa".



in modo tale che quando $x = k + 1$, nella seconda cella si trova $k = x - 1$, che deve essere trasferito nella cella iniziale. Poiché la funzione vale 0 quando $x = 0$, la prima cosa da fare è la seguente verifica $x \stackrel{?}{=} 0$; se la risposta è sì allora abbiamo terminato e possiamo uscire, altrimenti si continua col programma. Si tratta allora di porre il $+1$ nella terza cella e fare la verifica $r_1 \stackrel{?}{=} r_3$; se la condizione è soddisfatta si esce, altrimenti $k := k + 1$ e inizia un ciclo.

```
1        C(1, 4, 99)     (controlla se x = 0 usando la quarta cella)
2        S(3)            (pone +1 nella terza cella)
3   ┌──→ C(1, 3, 7)  ┆   (controlla se x = k + 1)
4   ┆    S(2)        ┆   (k := k + 1 nella seconda cella)
5   ┆    S(3)        ┆   (k := k + 1 nella terza cella)
6   └┄┄  C(1, 1, 3)  ┆   (salto incondizionato alla terza istruzione)
7        T(2, 1)   ←┄┄┘   (trasferisce il risultato x − 1 nella prima cella)
```

In figura 8.9 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso.

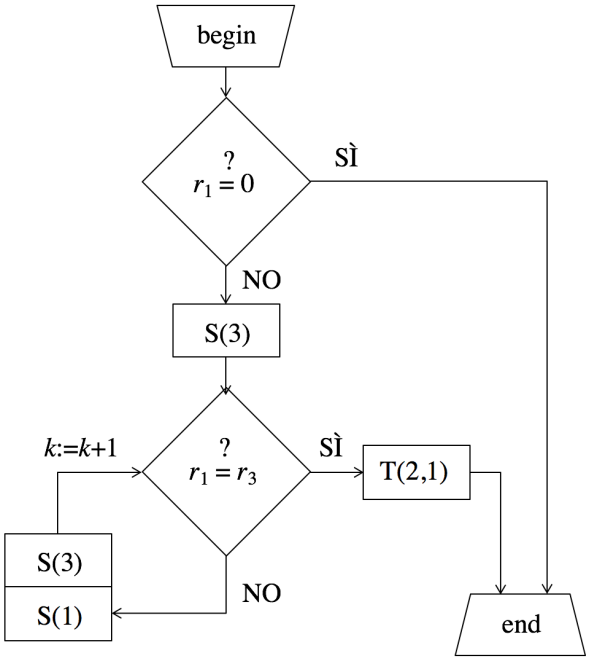


Figura 8.9: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

Esempio 8.6. Si voglia scrivere un programma che calcola la funzione $f(x) = x - y$ sui numeri naturali, che si definisce nel modo seguente

$$f(x) = x \dot{-} y = \begin{cases} x - y & x \geqslant y \\ \text{indef} & x < y \end{cases} \quad \forall (x, y)$$

Anche in questo caso si tratta di costruire una differenza a partire dalle quattro istruzioni di base, che *non* contemplano alcun tipo di sottrazione. Quando la funzione è definita si ha sempre $x \geqslant y$, e la differenza $x - y$

corrisponde al numero di volte in cui bisogna incrementare y per arrivare a x. La soluzione è allora quella di avere una configurazione corrente del tipo

x	$y+k$	k	0	0	$\cdots$

$y+k \stackrel{?}{=} x$

in modo tale che, quando $x = y + k$, nella terza cella si trova $k = x - y$, che deve essere poi trasferito nella cella iniziale. Anche in questa caso bisogna iniziare con un controllo, poiché potrebbe succedere che sia $x = y$, nel qual caso sarebbe già tutto finito.

```
1   ┌→  C(1,2,5)  ┄┐   (controlla se x = y + k)
2   ┆   S(2)       ┆   (k := k + 1 nella seconda cella)
3   ┆   S(3)       ┆   (k := k + 1 nella terza cella)
4   └┄  C(1,1,1)    ┆   (salto incondizionato alla prima istruzione)
5       T(3,1)  ←┄┘   (trasferisce il risultato x − y nella prima cella)
```

In figura 8.10 viene riportato il *diagramma a flusso* del programma, che mette in relazione le operazioni effettuate dallo stesso. Osserviamo che se $x < y$, sarà sempre vero che $x < y+k$, e quindi il confronto tra r_1 e r_2 della prima

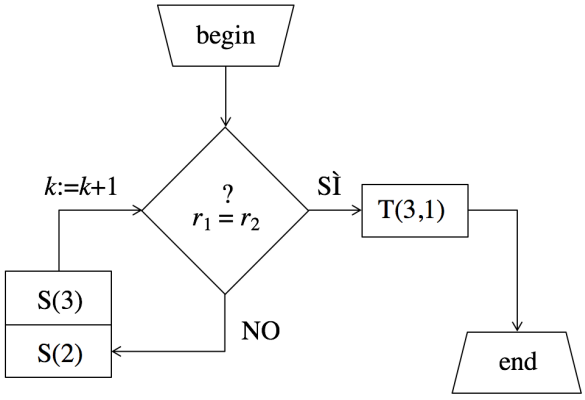


Figura 8.10: Diagramma a flusso del programma che calcola la funzione $f(x) = x \dot{-} 1$

istruzione $C(1, 2, 5)$ darà sempre esito negativo; non ci sarà quindi la possibilità di uscire dal ciclo per terminare con l'istruzione $T(3, 1)$ e la macchina ciclerà all'infinito. Ciò è coerente col fatto che per $x < y$ la funzione risulta non definita.

Esempio 8.7. Scriviamo un programma che calcola la funzione

$$f(x) = \begin{cases} x/2 & \text{se } x \text{ pari} \\ \text{indef} & \text{se } x \text{ dispari} \end{cases} \quad \forall x$$

La configurazione corrente che risolve il problema è $x, k, 2k, 0, 0, \ldots$, in modo tale che quando $x = 2k$ nella seconda cella si ha $k = x/2$. Ogniqualvolta incrementiamo k di un'unità nella seconda cella, dobbiamo incrementare di due unità la terza

```
1   ┌→  C(1,3,6)  ┄┐   (controlla se x = 2k)
2   ┆   S(2)       ┆   (k := k + 1 nella seconda cella)
3   ┆   S(3)       ┆   (k := k + 1 nella terza cella)
4   ┆   S(3)       ┆   (k := k + 2 nella terza cella)
5   └┄  C(1,1,1)    ┆   (salto incondizionato alla prima istruzione)
6       T(2,1)  ←┄┘   (trasferisce il risultato k = x/2 nella prima cella)
```

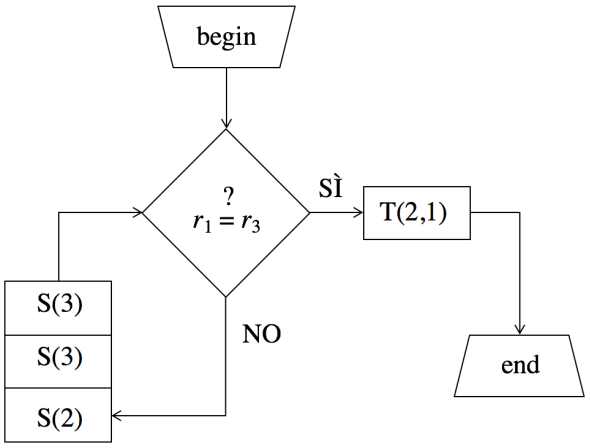


Figura 8.11: Diagramma a flusso del programma che calcola la funzione $f(x) = x/2$

Esempio 8.8. Scriviamo un programma che calcola le seguenti funzioni

$$f(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$S(1)$

$$f(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x \neq 0 \end{cases} \quad \forall x$$

1	$C(1,2,99)$
2	$Z(1)$
3	$S(1)$

Esempio 8.9. Scriviamo un programma che calcola le seguenti funzioni

$$f(x,y) = \begin{cases} 1 & \text{se } x = y \\ 0 & \text{se } x \neq y \end{cases} \quad \forall (x,y)$$

1	$C(1,2,4)$
2	$Z(1)$
3	$C(1,1,99)$
4	$Z(1)$
5	$S(1)$

$$f(x,y) = \begin{cases} 0 & \text{se } x = y \\ 1 & \text{se } x \neq y \end{cases} \quad \forall (x,y)$$

1	$C(1,2,5)$
2	$Z(1)$
3	$S(1)$
4	$C(1,1,99)$
5	$Z(1)$

Selezione e iterazione col linguaggio del modello RAM

Abbiamo più volte sottolineato il fatto che l'approccio procedurale-algoritmico fa riferimento al concetto di programma $\mathcal{P}$, costituito da un elenco di istruzioni $I_1, I_2, \ldots, I_s$, che devono essere eseguite in sequenza. Questo è l'approccio seguito nell'ambito della programmazione imperativa, dominante rispetto ad altri paradigmi. Ogni istruzione corrisponde a un "comando" che viene impartito alla macchina, e che prevede l'esecuzione di un certo lavoro. Nella sezione 7.1.1 abbiamo fissato le caratteristiche del *linguaggio macchina* che dà corpo a tali istruzioni; abbiamo visto che si tratta di istruzioni molto "povere", nel senso che dovendo essere direttamente eseguibili dal processore non possono prevedere elaborazioni troppo complesse, ma si limitano a operazioni di base del tipo "somma il contenuto di due celle di memoria", oppure "copia il contenuto di un cella in un'altra", "azzera il contenuto di una cella", ecc. Le istruzioni del linguaggio RAM sono proprio di questo tipo, e in tal senso costituiscono

un modello molto realistico del funzionamento del calcolatore a livello *hardware*.
Tuttavia, quando si deve realizzare un programma non si lavora mai a livello di linguaggio macchina, poiché sarebbe inutilmente faticoso, inefficiente e frustrante. Si preferisce invece operare a un livello logico superiore, usando linguaggi come C, C++, Java, Fortran, Pascal, ecc. per i quali le istruzioni elementari consentono di effettuare operazioni logiche più complesse di quanto si possa realizzare in linguaggio macchina, molto vicine alla logica che guida il ragionamento umano. In questo modo c'è anche il vantaggio che i programmi in tali linguaggi sono indipendenti dall'architettura della macchina. Ricordiamo inoltre che la traduzione tra un linguaggio ad *alto livello* di questo tipo e il linguaggio macchina, viene garantita da un programma che si chiama *compilatore*, e che andrà adattato alle diverse piattaforme architetturali in uso.
Se facciamo riferimento al paradigma più usato di programmazione imperativa, denominato *programmazione strutturata*, possiamo affermare che un programma è solitamente costruito nel seguente modo:

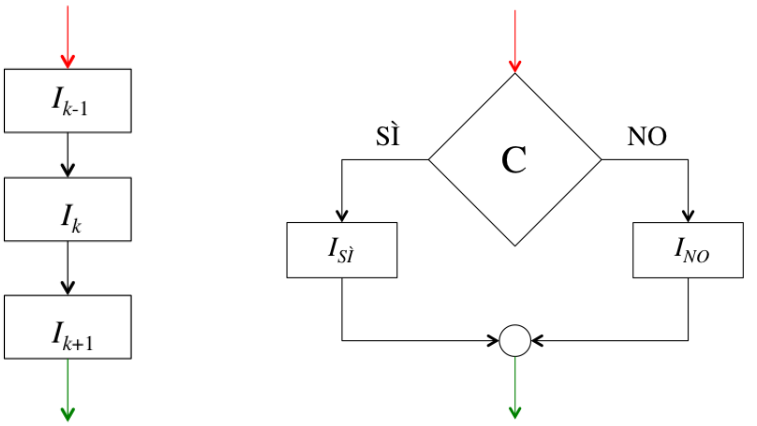
- *una parte dichiarativa*, in cui si dichiarano tutte le variabili del programma e il loro tipo (p.es. variabile intera, variabile carattere, ecc);

- *una parte che descrive l'algoritmo* risolutivo utilizzato, basato sulle istruzioni del linguaggio;
  a loro volta le istruzioni si dividono in:

  - istruzioni di lettura e scrittura (scrittura a video, scrittura su disco, lettura da tastiera, ...);
  - istruzioni di assegnamento (del valore a una variabile);
  - istruzioni logiche di controllo (frasi if, while, for, repeat, case, ...).

In un linguaggio strutturato, quale ad esempio il Pascal, ci sono sostanzialmente tre tipi di *strutture logiche di controllo* del programma; esse sono rispettivamente la *Sequenza*, la *Selezione* e l'*Iterazione*. Analizziamole nei dettagli:

Nella **Sequenza** (Fig. 8.12a) le istruzioni sono semplicemente poste in sequenza, una dopo l'altra. Il punto d'ingresso è evidenziato da una freccia rossa, quello di uscita da una freccia verde.



(a) Sequenza di istruzioni

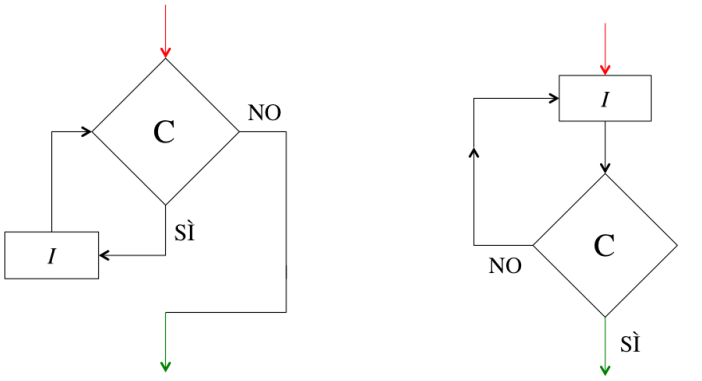
(b) Selezione effettuata sulla base della validità della condizione C; corrisponde a una frase Pascal **if** C **then** $I_{SÌ}$ **else** I_{NO}

Figura 8.12: Strutture logiche di controllo del tipo *sequenza* e *selezione*

Nella **Selezione** (Fig. 8.12b) si procede a una verifica della validità della condizione C, seguendo due strade diverse a seconda che C sia o meno vera; se C è vera (SÌ) si procede con l'esecuzione dell'istruzione $I_{SÌ}$, altrimenti (NO) si esegue l'istruzione I_{NO}. Nel linguaggio Pascal la selezione corrisponde alla frase **if** C **then** $I_{SÌ}$ **else** I_{NO}

Nell'**Iterazione** (Fig. 8.13) si attiva un *ciclo*, controllato dalla realizzazione di una certa condizione prefissata. Il

controllo può avvenire in due modi diversi: nel primo la validità della condizione C viene verificata subito; se essa è soddisfatta si esegue I altrimenti si esce. In questo caso si rimane nel ciclo e si continua a eseguire I finché la condizione vale; ciò corrisponde a una frase Pascal del tipo **while** C **do** I (si veda la figura 8.13a). Nel secondo caso viene eseguita subito l'istruzione I e solo successivamente si attua la verifica della condizione C; se essa non è soddisfatta si ri-esegue I altrimenti si esce; ciò corrisponde a una frase Pascal del tipo **repeat** I **until** C (si veda la figura 8.13b). Si osservi che in questo secondo caso l'istruzione I viene eseguita almeno una volta.



(a) Iterazione del tipo **while** C **do** I

(b) Iterazione del tipo **repeat** I **until** C

Figura 8.13: Due tipi diversi di iterazione

Immaginiamo ora che il linguaggio macchina del calcolatore sia basato sul linguaggio del modello RAM, e che si voglia "compilare", cioè tradurre una frase complessa del linguaggio Pascal in una sequenza di istruzioni RAM. Il punto nodale è chiaramente la costruzione delle strutture logiche di controllo, perché per il resto si tratta solo di assegnazioni o di operazioni di *input/output* dei dati. Si tratta quindi di costruire un programma RAM per ciascuna delle tre frasi **if-then-else**, **while-do** e **repeat-until**, in modo da poter dare le istruzioni corrette a livello macchina quando una di queste frasi venga evocata su un programma di un linguaggio ad alto livello. Cominciamo con la prima.

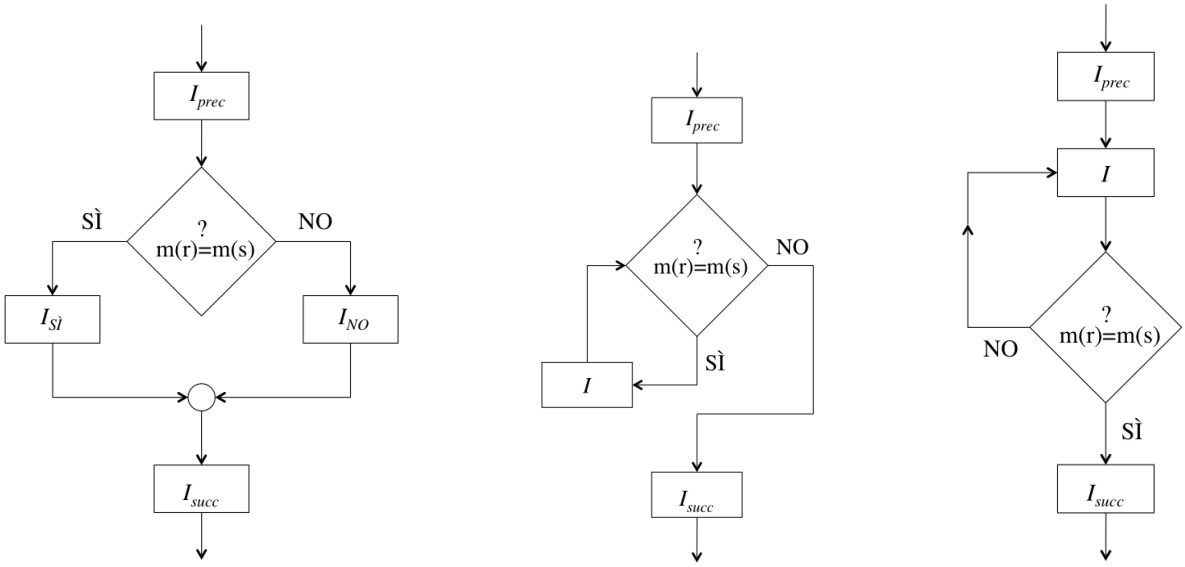
Frase di selezione **if** C **then** $I_{SÌ}$ **else** I_{NO}

Immaginiamo che la frase sia preceduta da una generica istruzione RAM che chiamiamo I_{prec} e seguita da (da una generica istruzione RAM che chiamiamo) I_{succ}. Se abbiamo già eseguito l'istruzione I_{prec} dobbiamo ora scegliere uno dei due percorsi, che portano a una delle due istruzioni $I_{SÌ}$ o I_{NO}, a seconda che la condizione C sia o meno vera. Per effettuare un tale tipo di verifica il linguaggio RAM ci offre l'istruzione $C(m, n, q)$; la condizione C di figura 8.12b deve essere dunque ricondotta a una verifica del tipo $r_m \stackrel{?}{=} r_n$. Se la risposta è sì si va a sinistra e si esegue $I_{SÌ}$; se la risposta è no si va a destra e si esegue I_{NO}. Sotto riportiamo le righe del codice associato.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m, n, k+3)$	(controlla se $r_m = r_n$)
$k+1$	I_{NO}	(se $r_m \neq r_n$ si esegue I_{NO})
$k+2$	$C(1, 1, k+4)$	(passa a I_{succ}, senza eseguire anche $I_{SÌ}$)
$k+3$	I_{SI}	(se $r_m = r_n$ si esegue $I_{SÌ}$)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **while** C **do** I

Eseguita I_{prec} si deve verificare la validità della condizione $r_m \stackrel{?}{=} r_n$. Se la condizione è soddifatta si innesca un ciclo che prevede l'esecuzione dell'istruzione I; tale istruzione non può tuttavia seguire immediatamente l'istru-



(a) Diagramma di flusso della frase **if-then-else**

(b) Diagramma di flusso della frase **while-do**

(c) Diagramma di flusso della frase **repeat-until**

Figura 8.14: Diagrammi di flusso delle frasi principali di selezione e di ciclo

zione di controllo, poiché l'istruzione immediatamente successiva a $C(m, n, .)$ è quella che viene attivata nel caso in cui la condizione *non* fosse stata soddisfatta; bisogna dunque fare un salto incondizionato $C(1, 1, k + 2)$ alla riga $k + 2$. Bisogna poi rifare la verifica $r_m = r_n?$, e per questo si attiva un salto incondizionato $C(1, 1, k)$, che ci riporta al passo k. Finché la condizione $r_m = r_n$ continua a essere soddisfatta si rimane nel ciclo. Nel caso invece la condizione non sia più soddisfatta si deve uscire dal ciclo, saltando direttamente all'istruzione successiva, di riga $k + 4$, con un salto incondizionato $C(1, 1, k + 4)$.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	$C(m, n, k+2)$	(controlla se $r_m = r_n$)
$k+1$	$C(1, 1, k+4)$	(se $r_m \neq r_n$ si deve andare a I_{succ})
$k+2$	I	(se $r_m = r_n$ si esegue I)
$k+3$	$C(1, 1, k)$	(salto incondizionato a I_k che chiude un ciclo)
$k+4$	I_{succ}	(esegue l'istruzione successiva)

Frase di iterazione **repeat** I **until** C

Subito dopo aver eseguito I_{prec} si esegue immediatamente anche la I. Poi si attua la verifica della condizione $r_m = r_n?$. Se la condizione non è soddisfatta si apre un ciclo con un'istruzione di salto incondizionato $C(1, 1, k)$, che ci riporta alla riga k; finché $r_m \neq r_n$ si rimane nel ciclo eseguendo più volte la I. Quando la C diventa vera si esce dal ciclo, andando all'istruzione successiva.

$k-1$	I_{prec}	(esegue l'istruzione precedente)
k	I	(esegue I)
$k+1$	$C(m, n, k+3)$	(se $r_m = r_n$ si va a I_{succ})
$k+2$	$C(1, 1, k)$	(se $r_m \neq r_n$ si apre un ciclo ritornando a I_k)
$k+3$	I_{succ}	(esegue l'istruzione successiva)

La costruzione dei programmi che sono il risultato della traduzione in linguaggio RAM delle tre frasi **if-then-else**, **while-do** e **repeat-until** ci dà la possibilità di creare il raccordo concettuale che esiste tra la scrittura di un programma ad alto livello e l'esecuzione delle corrispondenti istruzioni a seguito della compilazione nel linguaggio

macchina. Queste ultime, nel modello di calcolatore sotteso dal linguaggio RAM, occuperebbero le celle della memoria RAM illustrata sulla parte destra della figura 7.7, che descirive il ciclo *fetch-decode-execute* per l'esecuzione di un'istruzione di linguaggio macchina.

Bibliografia

[1] A. D'Amore, "I circuiti di commutazione", dispense.

[2] G. Bucci, "Calcolatori elettronici - Architettura e organizzazione", Mc Graw Hill Education, 2014.

[3] A. Clements, "Principle of computer hardware", Oxford University Press, 2005.

[4] J.L. Casti, W. De Pauli, "Gödel, A Life of Logic", Perseus Publishing.

[5] M. Davis, "Il Calcolatore Universale", Adelphi, Biblioteca Scientifica 35.

[6] M. Davis, "Is Mathematical Insight Algorithmic?",
http://cs.nyu.edu/cs/faculty/davism/penrose.ps

[7] M. Davis, "How Subtle is Gödel's Theorem",
http://cs.nyu.edu/cs/faculty/davism/penrose2.ps

[8] J.J. Gray, "The Hilbert Challenge", Oxford Univ. Press.

[9] F. Fabris, "Teoria dell'Informazione, codici, cifrari", Bollati Boringhieri, Torino, 2001.

[10] G.O. Longo, "Il nuovo Golem - Come il computer cambia la nostra cultura", Edizioni Laterza.

[11] Z. Manna, "Teoria Matematica della Computazione", Bollati Boringhieri, Torino, 1978.

[12] J. Roulston, "A Rough History of Computing",
www.gadae.com/news/Papers/A Rough History of Computing.pdf

[13] R. Spelta, www.storiainformatica.it

[14] www.wikipedia.org

Elenco delle figure