Dipartimento di Fisica "Enrico Fermi"
Corso di Laurea in Fisica

Introduzione all'Elettronica

Parte 1: Elettronica digitale

Elisa Falchini, Vincenzo Flaminio, Chiara Roda, Franco Spinella

Pisa, Settembre 2004

Avvertenze degli autori

Questo testo affronta un argomento che difficilmente potrebbe omettere il nome di compagnie che commercializzano software o hardware nel mondo dell'elettronica. Tutti i prodotti o le tecnologie a cui si fa riferimento sono marchi registrati a nome di queste compagnie.
Gli autori ringraziano anticipatamente i responsabili in questione e precisano che il materiale divulgato, anche se fa riferimento a marchi registrati, non ha carattere pubblicitario, né vuole favorire alcuno. Qualsiasi fraintendimento è da ritenersi di natura accidentale.

Prefazione

Il contenuto di questo volume riflette quello del Corso di "Laboratorio di Fisica V" tenuto da uno degli autori (V.F.) per gli studenti del terzo anno del Corso di Laurea in Fisica presso l'Università di Pisa.

L'impostazione che si è data è tale da non richiedere una conoscenza dell'elettronica analogica, argomento che nel corso specifico è trattato al secondo semestre e che sarà l'oggetto di un successivo volume. I soli presupposti, per lo studente che debba iniziare lo studio dell'elettronica digitale sono, oltre alle necessarie basi di matematica, una conoscenza dei concetti fondamentali dell'elettricità e dei circuiti elettrici.

Il volume è diviso in tre capitoli.

Nel primo, dopo una presentazione dei vari sistemi di numerazione adoperati, viene discussa l'algebra di Boole, che è alla base dei circuiti digitali, le porte logiche (AND, OR, NOT...), i metodi per implementare e semplificare le funzioni logiche, i circuiti di somma e sottrazione per numeri binari, i comparatori, i multiplexer e demultiplexer. Si affrontano poi gli elementi che sono alla base dei circuiti sequenziali, cioè i Flip-Flop e poi i registri, di cui vengono anche analizzate alcune applicazioni. Tra queste viene dato un certo rilievo ai generatori di sequenze casuali. Vengono infine descritti circuiti per realizzare operazioni di moltiplicazione e divisione tra numeri binari.

Nel secondo capitolo vengono discusse le strutture di alcuni tipi di memorie e poi le matrici logiche programmabili. Dato il grande sviluppo che in questi ultimi anni hanno avuto i "gate arrays" programmabili, abbiamo discusso con un certo dettaglio i Field Programmable Gate Arrays (FPGA).

Il terzo capitolo infine descrive sistemi logici di maggiore complessità, a partire dalle macchine a stati finiti. Vengono poi analizzati i microcontrollori/microprocessori, con particolare riguardo al PIC®16F877 della Microchip®. Vengono infine descritte le strutture logiche dei BUS, a partire dai BUS interni, per poi discutere i BUS di espansione e quelli di input/output.

Abbiamo cercato di chiarire quanto esposto con alcuni esempi ed applicazioni. Nel far ciò abbiamo fatto ricorso alla versione "Demo" del programma di simulazione MICROCAP 7 [1]. Questo è stato adoperato anche per ottenere molti dei grafici contenuti nel testo. Ringraziamo la Spectrum Software per il permesso di far uso di tale programma, il cui utilizzo (peraltro gratuito) consigliamo a tutti gli studenti.

Desideriamo ringraziare i numerosi Colleghi che hanno collaborato con noi in questi anni, contribuendo a sviluppare la parte pratica del Corso, assistendo gli studenti e mettendo inoltre in evidenza numerosi errori contenuti nelle prime stesure dei relativi "appunti". In particolare desideriamo ringraziare: Antonio Bardi, Alessan-

dro Cardini, Alessandro Cerri, Roberto Dell'Orso, Francesco Forti, Paola Giannetti, Enrico Mazzoni, Donato Nicolò e Sara Vecchio.

Un ringraziamento particolare, per la dedizione e la competenza con cui hanno gestito i laboratori didattici del secondo biennio di Fisica in tutti questi anni, al Responsabile dei laboratori, Carlo Bianchi, ed ai Tecnici Fabrizio Tellini e Virginio Merlin. Grazie al loro impegno il nostro lavoro è stato reso enormemente più agevole.

Indice

Capitolo 1

Sistemi Logici

1.1 Introduzione

Una delle prime esigenze che nel corso del processo di civilizzazione l'uomo ha incontrato, è stata quella di *contare* gli oggetti ed adoperare una qualche *rappresentazione grafica* per indicare il risultato di tale conteggio. Legata a questa è stata l'esigenza di *operare* su tali conteggi (sommare, sottrarre etc.).

I Romani, che pure avevano realizzato progressi impressionanti nel campo dell'Ingegneria, avevano adottato un sistema che, in analogia a sistemi adottati in altre antiche civiltà, associava un simbolo a ciascuno dei più importanti tra i numeri. Così: I=1, V=5, X=10, L=50, C=100, D=500, M=1000. Per rappresentare un numero generico si ricorreva ad una opportuna combinazione di tali simboli. Ad esempio il numero 27 si scriveva come: 10+10+5+2, ovvero: XXVII. Già in tale sistema aveva una certa importanza un concetto che è fondamentale nel sistema decimale: il *peso* attribuito alla *posizione* del simbolo all'interno della rappresentazione di un dato numero. Così: XI=11, ma IX=9.

Tale sistema non era certo il più adatto ad *operare* su numeri (sommare, moltiplicare etc.). Nell'antica Cina era stato invece sviluppato un sistema interamente *posizionale* ed un sistema di calcolo basato su tale sistema.

Tuttavia, il passo fondamentale nello sviluppo di un sistema che si prestava sia alla rappresentazione di numeri arbitrariamente grandi che ad operazioni su di essi fù compiuto in Egitto nel 3400 AC ed in Mesopotamia nel 3000 AC. Il passo consistette nell'associare al sistema posizionale, l'uso dello "zero".

Tale sistema fu importato in Occidente attorno al 1200 dal Pisano *Leonardo Fibonacci*, figlio di un mercante che viaggiava spesso nei paesi Africani e dell'Est. Egli, nel suo "Liber Abbaci" introdusse le cifre decimali ("novem figurae Yndorum") ed il segno "zephyrum" (lo zero).

Il sistema, noto oggi come "sistema Arabo", associava ad uno "0" posto alla destra di una cifra decimale (1-9) il significato di "moltiplicare per 10". Inoltre, una cifra decimale posta alla destra di un'altra cifra (o gruppo di cifre) implicava: "moltiplicazione per 10 seguita dalla somma del risultato alla cifra posta a destra". Così:

$$27 = 20 \times 10 + 7$$

La ragione per l'uso di 10 simboli diversi (nove più lo zero) era ovviamente legata all'associazione dei simboli con le dita delle mani (da cui la dizione Inglese "digit").

In un sistema elettronico digitale l'equivalente delle 10 dita sono i due stati in cui il sistema può trovarsi (transistor in conduzione o interdizione, tensione accesa o spenta etc.). I simboli divengono così 2, "0" ed "1". Il "digit" diventa ora il "bit".

1.2 Sistemi di numerazione

1.2.1 Il sistema binario

Nella vita quotidiana siamo abituati ad usare il sistema decimale. In tale sistema la "base" è 10, ciò che vuol dire l'uso di 10 simboli diversi per rappresentare i numeri da 0 a 9. I calcolatori elettronici, e con essi tutti i moderni sistemi di elettronica digitale, sono basati sull'uso di "celle di memoria" che possono memorizzare solo due simboli diversi, convenzionalmente chiamati 0 ed 1. Per vedere come, a partire da questi due soli simboli, sia possibile "costruire" un intero generico, torniamo un attimo al sistema decimale e vediamo di analizzare le operazioni mentali che istintivamente effettuiamo nello scrivere un numero intero decimale. Vediamo ciò con un esempio. Prendiamo il numero 23. Esso implica:

a scrivere la cifra 2

b moltiplicarla per 10

c aggiungere al risultato un 3

Se aggiungiamo al 23, a destra, un'ulteriore cifra, ad esempio un 7, ciò che facciamo è:

d moltiplicare il risultato delle operazioni a), b), c) precedenti per 10

e aggiungere al risultato un 7

Nel sistema adoperato nei calcolatori, detto sistema binario, il meccanismo adoperato per costruire un numero è il medesimo appena descritto, con la differenza che ora i simboli sono solo 0 ed 1 e che l'operazione "moltiplicazione per 10" diviene "moltiplicazione per 2". Vediamo un esempio. Prendiamo il numero 101. Effettuiamo le operazioni a), b), c), d), e) descritte, con la modifica menzionata:

a' scriviamo un 1

b' moltiplichiamo questo per 2

c' aggiungiamo al risultato la seconda cifra (0)

d' moltiplichiamo il risultato per 2

e' aggiungiamo al risultato la terza cifra

Quindi il numero binario scritto è, in decimale, un 5.

E' immediato vedere che, aggiungere al numero precedente uno 0 (a destra) equivale a moltiplicare per 2 il 5 ottenuto, ottenendo così un 10. Vediamo quindi che, come nel sistema decimale l'aggiunta di uno 0 a destra di un numero equivale

a moltiplicare questo per 10, così nel sistema binario l'aggiunta di uno 0 a destra di un numero equivale a moltiplicare questo per 2. Proviamo a convertire in decimale il seguente numero binario:

1 0 1 1 0 1 0 1 1

Seguiremo la procedura mostrata nella tabella 1.1:

1×2	$=$	2
$2 + 0$	$=$	2
2×2	$=$	4
$4 + 1$	$=$	5
5×2	$=$	10
$10 + 1$	$=$	11
11×2	$=$	22
$22 + 0$	$=$	22
22×2	$=$	44
$44 + 1$	$=$	45
45×2	$=$	90
$90 + 0$	$=$	90
90×2	$=$	180
$180 + 1$	$=$	181
181×2	$=$	362
$362 + 1$	$=$	363

Tabella 1.1: Procedura di trasformazione del numero binario 101101011 in un numero decimale.

Seguendo il procedimento indicato riusciamo a convertire in decimale, mentalmente ed in modo rapido, un numero binario non eccessivamente grande.

Possiamo, in modo equivalente, esprimere il numero decimale (o binario) dato, facendo uso del metodo delle potenze di 10 (o di 2). Così ad esempio, il numero decimale 314159 è esprimibile come mostrato nella tabella 1.2:

9×10^0	(9)	$+$	
5×10^1	(50)	$+$	
1×10^2	(100)	$+$	
4×10^3	(4000)	$+$	
1×10^4	(10000)	$+$	
3×10^5	(300000)	$=$	314159

Tabella 1.2: Esempio di un numero decimale espresso facendo uso delle potenze di 10.

Vale a dire che ogni cifra che compone il numero dato viene ad essere moltiplicata per una potenza di 10 pari alla posizione che essa occupa nel numero dato, contando

le posizioni da destra a sinistra ed attribuendo alla cifra posta all'estrema destra la posizione 0. Il numero è la somma di tutti i termini in tal modo ottenuti.

In modo analogo, nel sistema binario si scrive il dato numero come somma delle singole cifre binarie, moltiplicate ciascuna per una potenza di 2, corrispondente alla posizione che essa occupa nel numero. Ad esempio, il numero binario 100111010 si esprime come mostrato in tabella 1.3:

0×2^0	(0)	+	
1×2^1	(2)	+	
0×2^2	(0)	+	
1×2^3	(8)	+	
1×2^4	(16)	+	
1×2^5	(32)	+	
0×2^6	(0)	+	
0×2^7	(0)	+	
1×2^8	(256)	=	314

Tabella 1.3: Esempio di un numero binario espresso facendo uso delle potenze di 2.

Questo metodo, facile da implementare in un computer, è meno agevole per effettuare il calcolo mentalmente in modo rapido.

1.2.2 Il sistema ottale

Pur essendo il sistema binario quello adoperato nei sistemi digitali, altri sistemi diversi da quello binario sono spesso adoperati nello sviluppo del software che utilizza i sistemi digitali. I più comuni sono quello ottale (base 8) e quello esadecimale (base 16). Nel sistema ottale si dispone di 8 simboli (da 0 a 7) e si "costruiscono" i numeri maggiori di 7 seguendo i passi indicati ai punti a) ..... e), con la modifica del 10 in 8. Vediamo un esempio:

$$72_8 \ = \ 58_{10}$$

Cioè, come mostrato nella tabella 1.4:

7×8	=	56
$56 + 2$	=	58

Tabella 1.4: Procedura di trasformazione di un numero ottale in un decimale.

Ovviamente, in tale sistema il numero 8 si scriverà "10", cioè 1×8.

1.2.3 Il sistema esadecimale

Nel sistema esadecimale si dispone di 16 simboli (i numeri decimali da 0 a 9 e le lettere A, B, C, D, E, F). Abbiamo la situazione mostrata nella tabella 1.5:

A	$\rightarrow$	10_{10}
B	$\rightarrow$	11_{10}
C	$\rightarrow$	12_{10}
D	$\rightarrow$	13_{10}
E	$\rightarrow$	14_{10}
F	$\rightarrow$	15_{10}

Tabella 1.5: Corrispondenza tra le lettere che costituiscono parte del sistema esadecimale, e i numeri decimali.

I numeri maggiori di 15 vengono "costruiti" seguendo la ricetta delineata ai punti a) ..... e), con la sostituzione del 10 con il 16. Ad esempio il numero esadecimale "1A" in decimale è quello mostrato nella tabella 1.6:

1	$\times$	16	=	16
16	+	A	=	26

Tabella 1.6: Procedura di trasformazione di un numero esadecimale in uno decimale.

Il numero esadecimale "FF" è uguale a $(15 \times 16) + 15 = 255$.

1.2.4 Conversioni tra sistemi diversi

È facile convertire in ottale un numero binario. Basta tener presente che un insieme di tre bit binari può rappresentare i numeri base del sistema ottale, cioè i numeri tra 0 e 7. Possiamo allora raggruppare tre a tre i bit che costituiscono il numero binario dato (a partire dalla destra) per ottenere il numero in ottale. Così ad esempio:

$$[\underbrace{110}_{6}\underbrace{101}_{5}\underbrace{001}_{1}\underbrace{111}_{7}]_2 = [6517]_8$$

In modo analogo, è possibile convertire in esadecimale un generico numero binario. Questa volta si dovranno raggruppare quattro alla volta i bit che costituiscono il numero binario dato e sostituire a ciascun gruppo il corrispondente numero esadecimale. Ad esempio, il numero prima convertito in ottale, diventa:

$$[\underbrace{1101}_{D}\underbrace{0100}_{4}\underbrace{1111}_{F}]_2 = [D4F]_{16}$$

E' facile vedere poi come si possa agevolmente convertire un numero ottale o esadecimale in un numero binario. A tale scopo è sufficiente convertire in binario ciascuna delle cifre che costituiscono il numero ottale o esadecimale.

Ci si può chiedere ora come si possa convertire viceversa un numero decimale dato in una base 2. La procedura è la seguente:

(a) si divide per 2 il numero decimale assegnato, ottenendo un quoziente ed un resto. Notiamo che il resto sarà 0 (se il numero era pari) o 1 (se era dispari)

(b) si divide per 2 il quoziente ottenuto. Si ottiene un nuovo quoziente ed un nuovo resto

(c) si continua l'operazione, fino a che il quoziente non sia nullo

(d) la serie dei resti ottenuti, letta in ordine inverso, fornirà il numero binario richiesto.

Vediamo ciò con un esempio

$$[374]_{10} \;=\; [?]_2$$

Le successive operazioni sono quelle mostrate nella tabella 1.7.

Divisione	Quoziente	Resto
374/2	187	0
187/2	93	1
93/2	46	1
46/2	23	0
23/2	11	1
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

Tabella 1.7: Procedura di conversione di un numero decimale in un numero binario.

Il numero binario è quindi:

$$1\,0\,1\,1\,1\,0\,1\,1\,0$$

1.2.5 Numeri binari frazionari

Così come nel sistema decimale le cifre dopo la virgola decimale (o il punto decimale) vengono ad essere moltiplicate per potenza negative e crescenti di 10, così nel sistema binario le cifre dopo la virgola (binaria) vengono ad essere moltiplicate per potenze negative e crescenti di 2. Ad esempio:

$$[101.1101]_2 \;=\; [5.\left(\frac{1}{2}+\frac{1}{4}+\frac{0}{8}+\frac{1}{16}\right)]_{10}$$

Se vogliamo convertire un numero decimale frazionario in un numero binario, potremo procedere secondo le linee del seguente esempio.

Supponiamo di dover convertire il numero decimale 0.8125. Moltiplichiamo il numero per 2 e chiediamoci se il risultato è ≥ 1.

$$0.8125 \times 2 \;=\; 1.625 \quad \geq 1$$

Poiché la disuguaglianza è soddisfatta, scriviamo un 1 e sottraiamo un 1 dal risultato. Se non lo è scriveremo 0 e non effettueremo la sottrazione:

$$1.625 \; - \; 1 \; = \; 0.625 \Rightarrow \boxed{1}$$

Moltiplichiamo ora il numero ottenuto (0.625) per 2 e verifichiamo nuovamente se il risultato è ≥ 1. Se lo è scriviamo un altro 1 e sottraiamo 1 dal numero dato.

$$0.625 \times 2 \; = \; 1.25 \; \geq 1 \Rightarrow \boxed{1}$$

$$1.25 \; - \; 1 \; = \; 0.25$$

Moltiplichiamo ancora 0.25 per 2 e verifichiamo:

$$0.25 \times 2 \; = \; 0.5 \; \; < 1 \Rightarrow \boxed{0}$$

Poiché questa volta la disuguaglianza non è soddisfatta, abbiamo scritto uno zero anziché un 1.

Moltiplichiamo nuovamente per 2:

$$0.5 \times 2 \; = \; 1 \; \geq \; 1 \Rightarrow \boxed{1}$$

$$1 \; - \; 1 \; = \; 0 \quad \textit{fine del processo}$$

La sequenza di 1 e 0 scritta, letta dall'alto verso il basso, rappresenta il numero frazionario desiderato. Nel nostro caso esso è:

$$0.1101$$

Verifichiamo:

$$0.1101 \; = \; \frac{1}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} \; = \; 0.8125$$

1.2.6 Codifica di numeri binari in un computer

In una macchina a 32 bit, il più grande numero intero che è possibile rappresentare, tenendo conto del fatto che un bit è usato per il segno, è:

$$N \; = \; 2147483647$$

Si tratta di un numero grande, ma non abbastanza per tutte le situazioni. Per rappresentare numeri molto più grandi (o anche numero molto minori di 1) si ricorre alla rappresentazione che fa uso della caratteristica e della parte frazionaria. Ciò è in fondo analogo a quanto facciamo nel sistema decimale per rappresentare numeri molto grandi o molto piccoli, ricorrendo alla notazione esponenziale. Ad esempio, per il numero di Avogadro scriviamo:

$$N_A \; = \; 6.022045 \, 10^{23}$$

e per la costante di Boltzmann:

$$K \; = \; 1.380662 \, 10^{-23}$$

S	caratteristica (7 bits)		parte frazionaria (24 bits)	
0	1	7	8	31

Tabella 1.8: Rappresentazione standard dei numeri per il sistema IBM/370.

Così, nei computer, un numero può essere rappresentato dal prodotto di un numero binario per una potenza di 16, dove l'esponente è specificato da un secondo numero binario. Ad esempio, nel vecchio sistema IBM/370, un "parola" di 32 bit era divisa in due gruppi di bits, come mostrato nella tabella 1.8.

Il bit 0, all'estrema sinistra, è adoperato per il segno. I successivi 7 bit sono adoperati per la caratteristica. I rimanenti 24 costituiscono la parte frazionaria. Il punto decimale va immaginato alla sinistra del bit più significativo della parte frazionaria.

La caratteristica è ottenuta in realtà aggiungendo 64 all'esponente vero e proprio. Ciò vien fatto allo scopo di riuscire a rappresentare anche esponenti molto minori di zero.

Notiamo che i 7 bits della caratteristica possono rappresentare i numeri decimali che vanno da 0 a 127. Ad esempio, la rappresentazione di un numero che in notazione decimale fosse:

$$N_D \;=\; F_D \, 10^{C_D}$$

dove F_D è la parte frazionaria (.6022045 nel caso del numero di Avogadro) e C_D la caratteristica nello stesso esempio (cioè 24), si otterrebbe nel modo seguente:

a) si converte in binario la parte frazionaria, conservando solo 24 bits

b) si converte 10^{C_D} in 16^{C_H} , cioè in una potenza di 16

c) al valore di C_H ottenuto si somma 64 (decimale). Questa sarà la caratteristica del numero in esame.

La rappresentazione descritta è quella dei numeri detti "floating point" (ovvero, in italiano, "in virgola mobile") quando si adotti la "singola precisione". Per calcoli più accurati è possibile usare "parole" costituite da gruppi di 2 parole singole (ovvero 64 bit) o anche di 4 parole singole (ovvero 128 bit).

Poiché un gruppo di 8 bits costituisce quello che è chiamato un "byte", una parola "singola precisione" è costituita da 4 bytes, una in "doppia precisione" da 8 bytes, ed una in "precisione estesa" da 16 bytes.

1.2.7 Il codice Gray

Sempre restando nell'ambito di una logica di tipo binario, occorre notare che la costruzione che abbiamo descritto della corrispondenza tra numeri binari e numeri decimali, suggerita dall'analogia con il sistema decimale, non è unica. Infatti si può altrettanto facilmente ipotizzare una generica corrispondenza tra numeri binari e numeri decimali ed adottarla come "sistema di numerazione binaria". Tale corrispondenza deve essere implementabile tramite una precisa "ricetta". Ciò è fatto ad esempio dal codice Gray. In tale codice, i numeri binari 0 ed 1 corrispondono allo 0 ed 1 decimale rispettivamente, come mostrato nella tabella 1.9.

0000	→	0	
0001	→	1	
− − − −	− − −	− − − − −	prima riflessione
0011	→	2	
0010	→	3	
− − − −	− − −	− − − − −	seconda riflessione
0110	→	4	
0111	→	5	
0101	→	6	
0100	→	7	

Tabella 1.9: Regola di conversione tra i numeri "binari" del codice Gray e quelli decimali.

Per "costruire" la configurazione binaria corrispondente al 2 ed al 3 decimali, effettuiamo una "riflessione" delle prime due righe nello specchio rappresentato dalla prima linea tratteggiata e contemporaneamente cambiamo in 1 gli zeri che vengono a comparire nella seconda colonna a partire dalla destra. Per "costruire" ora la configurazione binaria corrispondente al 4, 5, 6, 7 decimali, effettuiamo una nuova riflessione delle prime quattro righe nello specchio rappresentato dalla seconda linea tratteggiata, e contemporaneamente cambiamo in 1 gli zeri che compaiono nella terza colonna, a partire dalla destra, delle quattro nuove righe.

Il processo può esser proseguito quanto si vuole. Notiamo che una caratteristica del sistema Gray è che nel passare da un numero al successivo, uno solo dei bit cambia. Così, nel passare dal 2 al 3 cambia solo il bit meno significativo, nel passare dal 3 al 4 cambia solo il terzo bit, etc. Ciò non accade nel sistema binario "convenzionale" dove, nel passaggio da un valore a quello adiacente possono cambiare un numero arbitrario di bits. Ciò è un indubbio vantaggio del codice Gray in applicazioni elettroniche, dove i tempi di commutazione degli elementi che memorizzano i diversi bit possono differire leggermente.

1.3 Porte logiche

L'informazione elementare alla base del funzionamento di un qualsiasi sistema digitale è il bit. I due valori che questo può acquistare, che convenzionalmente chiamiamo 0 ed 1, sono nella pratica associati a due possibili valori di tensione (ad esempio 0V e 5V) o di corrente. La convenzione che può essere adottata nello stabilire la corrispondenza tra stato logico (0 o 1) e livello di tensione (0V o 5V) stabilisce il tipo di logica. Se allo 0 logico facciamo corrispondere il più basso dei valori di tensione ed all'1 il più alto, diremo che stiamo lavorando in logica positiva. Viceversa, possiamo far corrispondere allo 0 logico il livello alto di tensione ed all'1 logico il livello basso; diremo che in tal caso stiamo lavorando in logica negativa.

Una variabile binaria può acquistare o il valore 0 o il valore 1. A partire da una o più variabili binarie possiamo poi, come vedremo tra un attimo, costruire delle

funzioni di questa/e variabile/i. Se una variabile, o funzione di variabili, acquista il valore 1, diremo anche che essa è vera (True = T). Se acquista il valore logico 0, diremo che è falsa (False = F). Ovviamente, come una variabile, così anche una funzione logica di variabili logiche può assumere solo i valori 1=T o 0=F.

La più semplice delle funzioni (logiche) di variabile logica è il NOT, o invertitore. Se A è la variabile, l'applicazione della funzione NOT fornisce una nuova variabile che è il complemento (logico) di A. Cioè, se A = 0, il NOT di A vale 1; se A = 1, il NOT vale 0. Il simbolo per indicare la negazione (cioè il NOT) è:

$$NOT(A) = \overline{A}$$

Il simbolo del circuito elettronico che implementa tale operazione è quello mostrato in figura 1.1:

Figura 1.1: Simbolo del circuito elettronico che implementa la funzione NOT.

Una funzione logica può esser caratterizzata scrivendo esplicitamente il valore assunto dalla funzione per ogni possibile valore del/degli ingressi. In genere si fa ciò sotto forma di una tabella, detta "tabella delle verità". Nel caso della funzione NOT questa è mostrata in figura 1.10:

A	$Y = \overline{A}$
0	1
1	0

Tabella 1.10: Tabella delle verità per la funzione NOT.

La funzione OR a due ingressi è definita dalla tabella delle verità 1.11.

A	B	$Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Tabella 1.11: Tabella delle verità per la funzione OR.

Cioè l'OR di A e B assume il valore 1 se almeno uno dei due ingressi A e B assume il valore 1. L'operatore che indica la funzione OR è il +:

Il simbolo del circuito logico che implementa la funzione OR è mostrato in figura 1.2:

La funzione AND a due ingressi è definita dalla tabella delle verità 1.12.

Vediamo dalla tabella che l'AND vale 1 se e solo se entrambi gli ingressi valgono 1. Il simbolo del circuito logico che implementa la funzione AND è mostrato in figura 1.3:

Figura 1.2: Simbolo del circuito elettronico che implementa la funzione OR.

A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Tabella 1.12: Tabella delle verità per la funzione AND.

Figura 1.3: Simbolo del circuito elettronico che implementa la funzione AND.

L'operatore che indica la funzione AND è quello della moltiplicazione: $A \cdot B$.

Porte OR ed AND a più di due ingressi sono possibili. L'uscita di una funzione OR a più ingressi vale 1 se almeno uno degli ingressi vale 1. L'uscita di una funzione AND a più ingressi vale 1 se tutti gli ingressi valgono 1.

E' ovviamente possibile combinare due o più funzioni logiche per definire nuove funzioni. Così ad esempio, un AND seguito da un NOT realizza la funzione nota come NAND (= NOT·AND). La tabella delle verità di un NAND è quella data nella 1.13

A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Tabella 1.13: Tabella delle verità della funzione NAND.

Il simbolo del circuito elettronico che implementa la porta NAND è mostrato in figura 1.4:

Figura 1.4: Simbolo del circuito elettronico che implementa la funzione NAND.

Un OR seguito da un NOT realizza la funzione nota come NOR (= NOT·OR), la cui tabella delle verità è la 1.14

Il simbolo del circuito elettronico che implementa la porta NOR è mostrato in figura 1.5:

La funzione "OR esclusivo", indicata con XOR, ha la medesima tabella delle verità della funzione OR, ad eccezione della riga con A = B = 1, in cui l'XOR vale

A	B	$Y = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Tabella 1.14: Tabella delle verità della funzione NOR.

Figura 1.5: Simbolo del circuito elettronico che implementa la funzione NOR.

0. Possiamo pensare all'XOR come un "OR con esclusione dell'AND". Infatti la riga con A = B = 1 è quella per la quale è soddisfatto l'AND di A e B. Il simbolo del circuito che implementa la funzione XOR è mostrato in figura 1.6:

Figura 1.6: Simbolo del circuito elettronico che implementa la funzione XOR.

Il simbolo per l'XOR è il $\oplus$.

Sono disponibili in commercio integrati che effettuano le operazioni elementari descritte: AND, OR, NOT, NAND, XOR etc. I relativi chip sono distinti da sigle che variano a seconda della casa produttrice e che specificano caratteristiche diverse del relativo dispositivo. Esiste tuttavia una convenzione in base alla quale le ultime due cifre numeriche nella sigla indicano il tipo di funzione eseguita dall'integrato. Ad esempio, la sigla SN74LS00 indica un chip che esegue la funzione NAND a 2 ingressi. Le cifre numeriche che indicano la funzione NAND sono le ultime due (00). Più in particolare si tratta di quattro porte NAND realizzate sul medesimo chip ($QUAD$ 2 input NAND gate). Analogamente, il codice 02 indica una porta NOR etc. Alcuni altri chip standard disponibili in commercio sono elencati nella tabella 1.15.

1.3.1 Caratteristiche delle porte logiche

Le descrizioni delle porte logiche che abbiamo dato nella sezione precedente sono alquanto idealizzate ed è necessario a questo punto fornire alcune precisazioni.

Prendiamo ad esempio il caso di una porta NOT. Se l'ingresso alla porta vale 0, l'uscita vale 1. Nel caso particolare in cui la porta sia di tipo TTL[1], con livelli di tensione di 0 V e +5 V, quando l'ingresso è a 0 V, l'uscita sarà vicina a +5 V, viceversa, quando l'ingresso sarà vicino a +5 V, l'uscita sarà prossima a 0 V. L'andamento realmente osservato sarà, qualitativamente, quello mostrato in figura 1.7.

La transizione dell'ingresso da 0 V a +5 V non è istantanea, ma avviene con un andamento caratterizzato da un "tempo di salita". Analogamente, il segnale non tornerà istantaneamente a 0 V, ma lo farà con un andamento caratterizzato da un certo "tempo di discesa". L'uscita avrà un andamento analogo (rovesciato)

[1]TTL sta per Transistor-Transistor Logic.

Dispositivo	Funzione
00	Quad 2 input NAND
02	Quad 2 input NOR
04	Hex (6 in un chip) NOT
08	Quad 2 input AND
10	Triple 3 input NAND
20	Dual 4 input NAND
27	Triple 2 input NOR
30	8 input NAND
32	Quad 2 input OR
86	Quad exclusive OR
135	Quad exclusive OR/NOR
136	Quad exclusive OR

Tabella 1.15: Convenzione per l'identificazione delle funzioni logiche implementate dai chip disponibili in commercio.

Figura 1.7: Segnali di input ed output di una porta logica NOT, con relativi tempi di ritardo e propagazione.

ma le transizioni che la caratterizzano avverranno con un certo ritardo. Si definisce ritardo di propagazione (propagation delay) il tempo che intercorre tra l'istante in cui l'ingresso passa per un valore di tensione pari al 50% del suo valore massimo ed il corrispondente istante in cui l'uscita passa per un valore di tensione pari al 50% del suo valore massimo.

Come si vede dalla figura, abbiamo in realtà due diversi tempi di propagazione: il primo di questi caratterizza la transizione dell'ingresso dal valore basso (L) a quello alto (H). Questo ritardo è noto con tp_{HL}. L'altro tempo di propagazione, relativo al passaggio dell'ingresso dal valore alto a quello basso, è indicato con tp_{LH}. Il primo di questi ritardi di propagazione vale tipicamente 20 ns, il secondo 15 ns, per un invertitore standard TTL.

Analoghi ritardi di propagazione caratterizzano tutte le porte logiche esaminate, nonché i circuiti logici che discuteremo in seguito. Per ciascun integrato la casa produttrice fornisce i valori tipici di tali ritardi.

Una discussione dettagliata di questi aspetti, come pure delle variazioni che i livelli di tensione considerati $High$ e Low possono subire e delle relative tolleranze, sarà fatta nel secondo volume.

1.4 Algebra di Boole

1.4.1 Identità Booleane

A partire da due o più variabili logiche A, B, C etc., è possibile costruire espressioni logiche di varia complessità. Tali espressioni possono essere semplificate facendo uso delle regole dell'algebra di Boole. Tali regole sono nella più gran parte dei casi immediatamente ovvie; in altri possono essere dimostrate costruendo la relativa tabella delle verità per evidenziare l'uguaglianza dei due membri dell'equazione.

Elenchiamo le più frequenti:

1. $A + B + C = (A + B) + C = A + (B + C)$ (proprietà associativa)
2. $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$ (proprietà associativa)
3. $A + B = B + A$ (proprietà commutativa)
4. $A \cdot B = B \cdot A$ (proprietà commutativa)
5. $A + A = A$ (identità)
6. $A \cdot A = A$ (identità)
7. $A + 1 = 1$
8. $A + 0 = A$
9. $A \cdot 1 = A$
10. $A \cdot 0 = 0$
11. $A \cdot (B + C) = A \cdot B + A \cdot C$ (proprietà distributiva)
12. $A + (A \cdot B) = A$ (ridondanza)
13. $A \cdot (A + B) = A$ (ridondanza)
14. $A + (B \cdot C) = (A + B) \cdot (A + C)$
15. $\overline{\overline{A}} = A$
16. $A \cdot \overline{A} = 0$
17. $A + \overline{A} = 1$
18. $A + (\overline{A} \cdot B) = A + B$
19. $A \cdot (\overline{A} + B) = A \cdot B$

A	B	C	$(B \cdot C)$	$A + (B \cdot C)$	$(A + B)$	$(A + C)$	$(A + B) \cdot (A + C)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Tabella 1.16: Dimostrazione della regola (14) dell'algebra di Boole.

$\overline{A}$	$\overline{B}$	A	B	$A \cdot B$	$\overline{A \cdot B} = \overline{A} + \overline{B}$
1	1	0	0	0	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	1	0

Tabella 1.17: Verifica dell'uguaglianza tra un AND in logica positiva ed un OR in logica negativa.

Dimostriamo, facendo uso del metodo della tabella della verità, la (14). Il procedimento è mostrato nella tabella 1.16.

Notiamo che se A = 1 in logica positiva, allora A = 0 in logica negativa. Consideriamo ora la tabellina dell'AND (in logica positiva) (tabella 1.17).

Cambiare logica vuol dire complementare tutto, cioè scambiare 0 con 1 nella tabellina. Abbiamo indicato ciò nelle prime due colonne e nell'ultima. Notiamo che l'ultima colonna coincide con l'OR delle prime due. Possiamo allora dire che: "un AND in logica positiva diviene un OR in logica negativa". Questo risultato è anche conseguenza delle leggi di De Morgan, che esamineremo tra poco.

Esaminiamo ora la tabella delle verità (tabella 1.18) della funzione XOR ($Y = A \oplus B$).

A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 1.18: Tabella delle verità della funzione XOR.

La tabella ci permette di scrivere diverse espressioni della funzione XOR, facendo

uso di funzioni OR, NOT ed AND. Una prima espressione si può ottenere esaminando le due righe in corrispondenza delle quali è Y = 1. La prima di queste corrisponde a A falso e B vero, cioè all'AND: $\overline{A} \cdot B$; la seconda corrisponde ad $A \cdot \overline{B}$. Poiché Y vale 1 sia nel primo caso che nel secondo, potremo scrivere:

$$1)\ A \oplus B \ = \ \overline{A} \cdot B + A \cdot \overline{B} \tag{1.1}$$

Una seconda espressione dell'XOR può essere ottenuta notando che l'XOR è vero quando è vero l'OR a condizione che sia falso l'AND. Cioè:

$$2)\ A \oplus B \ = \ (A + B) \cdot (\overline{A \cdot B}) \tag{1.2}$$

Una terza espressione si può ottenere notando che l'XOR è vero se è vero A o B: $(A + B)$ purché uno dei due sia falso $(\overline{A} + \overline{B})$:

$$3)\ A \oplus B \ = \ (A + B) \cdot (\overline{A} + \overline{B}) \tag{1.3}$$

Infine notiamo che il complemento dell'XOR è vero (cioè l'XOR è falso) nella prima e quarta riga della tabella delle verità.

Queste corrispondono a $\overline{A} \cdot \overline{B}$ (la prima) e a $A \cdot B$ l'ultima. Quindi:

$$\overline{A \oplus B} \ = \ \overline{A} \cdot \overline{B} + A \cdot B$$

cioè:

$$4)\ A \oplus B \ = \ \overline{\overline{A} \cdot \overline{B} + A \cdot B} \tag{1.4}$$

L'implementazione dei quattro modi indicati per realizzare la funzione XOR è indicata nelle figure 1.8, 1.9, 1.10 ed 1.11.

Figura 1.8: Primo metodo per l'implementazione della funzione XOR.

Figura 1.9: Secondo metodo per l'implementazione della funzione XOR.

Figura 1.10: Terzo metodo per l'implementazione della funzione XOR.

Figura 1.11: Quarto metodo per l'implementazione della funzione XOR.

1.5 Leggi di De Morgan

L'equivalenza delle quattro diverse implementazioni della funzione XOR, mostrata nella sezione precedente, può esser dimostrata in modo rigoroso facendo uso delle leggi di De Morgan. Esse sono:

$$\overline{A \cdot B \cdot C \cdots} = \overline{A} + \overline{B} + \overline{C} + \cdots$$

$$\overline{A + B + C + \cdots} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdots$$

La prima di queste ci dice che il complemento dell'AND di più variabili logiche è uguale all'OR dei complementi delle singole variabili. La seconda dice che il complemento dell'OR di più variabili logiche è uguale all'AND dei complementi delle singole variabili. Le leggi di De Morgan sono utilissime nel semplificare o trasformare espressioni logiche in più variabili.

Vediamo ora come, facendo uso di tali leggi, si possa facilmente dimostrare l'equivalenza delle quattro diverse espressioni che abbiamo fornito per la funzione XOR. Partiamo ad esempio dall'espressione 1.1:

$$A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

Complementandola due volte, otteniamo:

$$\cdots \overline{\overline{\overline{A} \cdot B + A \cdot \overline{B}}} = \overline{\overline{\overline{A} \cdot B} \cdot \overline{A \cdot \overline{B}}} = \overline{(A + \overline{B}) \cdot (B + \overline{A})} = \overline{A \cdot B + \overline{A} \cdot \overline{B}}$$

ciò troviamo la 1.4 data in precedenza.

Sviluppando quest'ultima troviamo poi:

$$\overline{A \cdot B + \overline{A} \cdot \overline{B}} = \overline{A \cdot B} \cdot \overline{\overline{A} \cdot \overline{B}} = (\overline{A} + \overline{B}) \cdot (A + B)$$

cioè la forma 1.3 dell'XOR.

Infine, complementando due volte il primo termine nell'ultimo membro dell'espressione appena scritta:

$$\overline{\overline{(\overline{A} + \overline{B})}} \cdot (A + B) = (\overline{A \cdot B}) \cdot (A + B)$$

che è la forma 1.2 dell'XOR. E' così dimostrata l'equivalenza delle quattro forme.

Facendo uso dell'algebra di Boole e delle leggi di De Morgan è possibile semplificare proposizioni logiche complesse. E' inoltre possibile ottenere delle realizzazioni concrete di funzioni logiche, facendo uso di pochi componenti base. Ad esempio, è facile vedere come la funzione OR possa esser ottenuta facendo uso di sole porte NAND. Consideriamo infatti un OR con due ingressi A e B:

$$A + B$$

Negando due volte otteniamo:

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

cioè il simbolo circuitale mostrato in figura 1.12, che usa solo porte NAND.

Figura 1.12: Implementazione della funzione OR facendo uso solo delle porte NAND.

1.6 Forme standard di funzioni logiche

Nel seguito, per semplificare la notazione, ometteremo il simbolo $\cdot$ che indica il prodotto logico di due variabili.

Esistono due diverse forme standard per le funzioni logiche. Esse sono rispettivamente la *somma di prodotti* ed il *prodotto di somme*. Un esempio del primo tipo di funzione è:

$$XY + \overline{W}X + \overline{W}YZ + XYZ$$

dove compaiono le quattro variabili X,Y,Z,W.

Facciamo vedere ora come questa possa essere riscritta nella forma di un *prodotto di somme*. L'espressione può essere riscritta nella forma:

$$\begin{aligned} XYX + \overline{W}X + \overline{W}YZ + XYYZ = \\ (\overline{W} + XY)X + (\overline{W} + XY)YZ = \\ (\overline{W} + XY)(X + YZ) \end{aligned}$$

che è un prodotto di somme.

Si dice che una somma di prodotti è in forma estesa se ciascun termine della somma contiene tutte le variabili. Tale forma emerge in modo naturale se la funzione logica è ottenuta da una tabella delle verità. Vediamo ciò con un esempio (tabella 1.19).

X	Y	Z	L
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Tabella 1.19: Somma di prodotti in forma estesa delle variabili X, Y e Z.

Dalla tabella di vede subito che:

$$L = \overline{X}Y\overline{Z} + \overline{X}YZ + X\overline{Y}\overline{Z} + XY\overline{Z} + XYZ$$

Ciascuno dei termini della somma va sotto il nome di "minterm". Notiamo che l'espressione scritta può esser semplificata:

$$L = \overline{X}Y(\overline{Z} + Z) + X\overline{Y}\overline{Z} + XY(Z + \overline{Z}) = \overline{X}Y + X\overline{Y}\overline{Z} + XY =$$

$$= Y(X + \overline{X}) + X\overline{Y}\overline{Z} \;=\; Y + \overline{Y}X\overline{Z} \;=\; Y + X\overline{Z}$$

Un esempio di prodotto di somme è:

$$L \;=\; (\overline{W} + XY)(X + YZ)$$

Ciascuno dei termini del prodotto è un "maxterm". La funzione scritta può facilmente esser trasformata in una nuova espressione dove ciascun termine del prodotto contiene solo due delle variabili. Facendo uso della relazione:

$$A + (BC) \;=\; (A + B) \cdot (A + C)$$

troviamo:

$$L \;=\; (\overline{W} + X)(\overline{W} + Y)(X + Y)(X + Z)$$

Diremo che il prodotto di somma è in *forma estesa* se ciascuna somma contiene tutte le variabili. Tale forma può facilmente essere ottenuta nel caso in cui la funzione L sia definita da una tabella della verità. Ad esempio, facendo uso della medesima tabella usata sopra, calcoliamo $\overline{L}$:

$$\overline{L} \;=\; \overline{XYZ} + \overline{XY}Z + X\overline{Y}Z$$

da cui:

$$\overline{\overline{L}} \;=\; L \;=\; \overline{\overline{XYZ} + \overline{XY}Z + X\overline{Y}Z} \;=\; \overline{\overline{XYZ}} \cdot \overline{\overline{XY}Z} \cdot X\overline{Y}Z =$$

$$(X + Y + Z)(X + Y + \overline{Z})(\overline{X}Y\overline{Z})$$

che è un prodotto di somme in forma estesa.

1.7 Mappe di Karnaugh

Nel progettare un circuito che implementi una certa funzione logica, si parte dalla definizione della funzione, analizzando tutte le possibili combinazioni delle variabili logiche d'ingresso in corrispondenza alle quali essa assume il livello logico 1. Questo è certamente vero quando la funzione è espressa tramite i *minterm*, cioè come somma di prodotti. Se la funzione è espressa come un prodotto di somme, cioè tramite i *maxterm*, è sempre possibile trasformarla in una somma di prodotti.

È alternativamente possibile sviluppare un formalismo analogo a quello che ora discuteremo, per funzioni espresse in termine dei maxterm. Per dettagli si consulti la referenza [3].

Si ottiene in generale una struttura contenente combinazioni di AND, OR, NAND etc.. Non è però detto che la struttura così progettata sia la più economica possibile. Altre strutture equivalenti possono esser trovate che minimizzano il numero dei componenti necessari e quindi il costo, la compattezza e la velocità.

Vediamo ciò con un semplice esempio. Si debba realizzare un circuito con due ingressi A e B ed un'uscita Y, definita dalla tabella delle verità 1.20.

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	0

Tabella 1.20: Tabella delle verità per un circuito a due ingressi ed un'uscita definita da $Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B}$.

Questa è esprimibile nella forma:

$$Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B}$$

che richiederebbe due porte AND, due NOT ed un OR. Tale relazione può però esser facilmente semplificata:

$$Y = (\overline{A} + A) \cdot \overline{B} = \overline{B}$$

Vediamo così che è sufficiente far uso di una singola porta NOT.

Il caso esaminato è di estrema semplicità e siamo quindi riusciti a minimizzare l'espressione facendo uso soltanto delle leggi dell'algebra di Boole. In casi più complessi (ma purché il numero di variabili sia limitato) si può far ricorso ad un metodo grafico, noto come "metodo delle mappe di Karnaugh".

Una mappa di Karnaugh (o semplicemente mappa K) è una matrice di celle che fornisce una rappresentazione visiva immediata della tabella delle verità.

Iniziamo con l'illustrare tale metodo esaminando la sua applicazione al semplice problema che abbiamo appena visto. Riportiamo su di un asse i possibili valori della variabile A e su di un asse ortogonale quelli della variabile B, come mostrato in figura 1.13.

Figura 1.13: Mappa di Karnaugh per la tabella delle verità 1.20.

Le celle contenenti "1" o "0" corrispondono alle combinazioni di valori di A e B in corrispondenza alle quali la Y è rispettivamente vera (1) o falsa (0). Nella pratica si indicano solo gli "1", lasciando vuote le celle corrispondenti agli "0".

La prima delle regole relative alle mappe K ci dice che due caselle adiacenti (lungo una riga o una colonna) contenenti "1" possono esser accorpate, come indicato in figura, essendo evidente che la Y vale "1" in corrispondenza al valore della variabile d'ingresso che è uguale nelle due caselle. Nel nostro esempio ciò significa che la Y è *vera* se B è $falso$:

$$Y = \overline{B}$$

In generale, nel caso di una funzione di due sole variabili, due caselle adiacenti in cui è presente un "1" danno luogo ad un termine che contiene la singola variabile che è *vera* o *falsa* nelle due caselle. Se viceversa è presente un "1" isolato, questo darà luogo ad un termine contenente entrambe le variabili.

Esaminiamo ancora un esempio, dato dall'espressione:

$$Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B} + A \cdot B$$

Questa è descritta dalla mappa K di figura 1.14.

Figura 1.14: Mappa di Karnaugh per l'espressione $Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B} + A \cdot B$.

Qui abbiamo due coppie di "1" adiacenti, come indicato dalle curve in figura. La prima di queste (prima riga) corrisponde al termine $\overline{B}$, la seconda (seconda colonna) ad A. Avremo quindi:

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

come d'altronde è facile verificare facendo uso dell'algebra di Boole.

Esaminiamo ora un caso in cui le variabili d'ingresso siano tre: A, B, C:

$$Y = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C + \overline{A} \cdot B \cdot \overline{C} + A \cdot B \cdot \overline{C}$$

In tal caso conviene riportare su di un asse (ad esempio quello orizzontale) tutte le possibili combinazioni delle prime due variabili e sull'altro i possibili valori della terza. Nel riportare le prime due si procederà in ordine tale che le due variabili, nel passare da una cella a quella adiacente, cambino una alla volta, cioè la sequenza sarà: 00,01,11,10. La matrice K che così si ottiene è del tipo mostrato in figura 1.15

Figura 1.15: Mappa di Karnaugh per un caso con tre variabili in ingresso.

È importante notare che la mappa và sempre immaginata *estesa*, nel senso che, nel caso specifico, occorre immaginare che esistano, a destra ed a sinistra di quella data, due mappe identiche ad essa, i cui elementi contenenti "1" possono esser combinati con identici elementi adiacenti presenti nella mappa originale. Le estensioni sono indicate in figura dalle matrici tratteggiate (dove non abbiamo riportato in modo esplicito gli "1" presenti, che vanno sottintesi).

Nell'esempio di figura vediamo un raggruppamento di quattro celle (indicato dalla linea tratteggiata) che corrisponde, come si può facilmente verificare, a:

$$Y = \overline{A}$$

Abbiamo inoltre un ulteriore raggruppamento di due caselle, mostrato in figura, corrispondente a:

$$Y = B \cdot \overline{C}$$

Si ha in definitiva:

$$Y = \overline{A} + B \cdot \overline{C}$$

Esaminiamo ora le regole generali per la semplificazione di funzioni logiche di tre variabili, come quella appena esaminata:

1. Quattro celle adiacenti (in linea o quadrato) si combinano in modo da fornire una funzione di una sola variabile.
2. Due celle adiacenti si combinano in modo da fornire una funzione di due variabili.
3. Una cella isolata dà luogo ad un termine funzione di tutte e tre le variabili.

Esaminiamo ancora un esempio di sistema dipendente da tre variabili. Si voglia realizzare un circuito che soddisfi la tabella delle verità 1.21:

A	B	C	Y_1	Y_2	Y_3
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	0	0	0

Tabella 1.21: Esempio di tabella delle verità con tre variabili in ingresso.

In figura 1.16 mostriamo le mappe K per ciascuna delle funzioni Y_1, Y_2, Y_3.

Figura 1.16: Mappe di Karnaugh per ciascuno degli esempi della tabella 1.21.

Vediamo facilmente, applicando le regole date che:

$$Y_1 = \overline{A} \cdot \overline{B} + A \cdot \overline{C}$$

$$Y_2 = \overline{C} + \overline{A}$$

$$Y_3 = \overline{B} + \overline{A}$$

dove nel caso di Y_3 si vede che la prima e l'ultima colonna si combinano a dare il termine $\overline{B}$.

Passiamo infine al caso di una mappa a quattro variabili, come quella di figura 1.17.

Figura 1.17: Mappa di Karnaugh per un sistema a quattro variabili in ingresso.

Anche in questo caso occorre ricordare che la mappa è da immaginare *estesa*, questa volta in tutte le direzioni, come schematizzato in figura 1.18.

Le regole da ricordare, come si può facilmente verificare, sono adesso:

Figura 1.18: Schematizzazione *estesa* della mappa di Karnaugh di figura 1.17.

1. Un rettangolo di otto celle adiacenti dà luogo ad un termine in una singola variabile;

2. Quattro celle adiacenti (in linea o quadrato) danno luogo ad un termine in due variabili;

3. Due celle adiacenti danno luogo ad un termine contenente tre variabili;

4. Una cella singola corrisponde ad un termine contenente tutte e quattro le variabili.

Esaminiamo ad esempio l'espressione:

$$Y = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} + A \cdot B \cdot \overline{C} \cdot \overline{D} + A \cdot B \cdot \overline{C} \cdot D + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D +$$

$$+ \overline{A} \cdot B \cdot \overline{C} \cdot \overline{D} + \overline{A} \cdot B \cdot C \cdot \overline{D} + A \cdot B \cdot C \cdot \overline{D} + A \cdot B \cdot C \cdot D$$

La corrispondente mappa K è mostrata in figura 1.19.

Figura 1.19: Mappa di Karnaugh per l'esempio considerato, a quattro variabili.

Che, come si può facilmente verificare, fornisce l'espressione semplificata:

$$Y = \overline{A} \cdot \overline{B} \cdot \overline{C} + B \cdot \overline{D} + A \cdot B$$

Consideriamo ancora un esempio. Facendo uso di tre motori si vuol comandare la rotazione e lo spostamento della punta di un trapano nel piano x-y, nel modo seguente:

(a) Un deviatore P comanda l'accensione (P=1) e lo spegnimento (P=0) del motore principale MP, che provoca la rotazione della punta

(b) Due deviatori di direzione Dx e Dy comandano l'accensione (Dx=1 o Dy=1) e lo spegnimento (Dx=0 o Dy=0) di due motori Mx ed My che fanno spostare la punta nelle due direzioni x ed y

(c) Un comando I seleziona i due versi per il movimento (I=0 in avanti; I=1 indietro)

(d) Un pulsante di allarme A, se premuto, spegne tutti i motori (A=1, altrimenti A=0)

(e) Il motore My deve essere spento (My=0) se il motore principale MP è acceso

P	Dx	Dy	I	MP	Mx	My
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	0
1	0	1	1	1	0	0
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	0
1	1	1	1	1	1	0

Tabella 1.22: Tabella delle verità relativa all'esempio per il controllo della punta di un trapano.

Figura 1.20: Mappa di Karnaugh per la variabile MP.

Figura 1.21: Mappa di Karnaugh per la variabile Mx.

Figura 1.22: Mappa di Karnaugh per la variabile My.

Come si vede, abbiamo cinque variabili (P, Dx, Dy, I ed A). Per ridurci a sole quattro variabili, eliminiamo la considerazione del caso ovvio in cui il pulsante d'allarme è stato premuto, per cui tutti i motori sono spenti. Ammettiamo quindi che sia sempre A=0.

La tabella delle verità è allora quella mostrata nella tabella 1.22.

Le mappe K per MP, Mx, My sono mostrate nelle figure 1.20, 1.21, 1.22.

Da queste otteniamo:

$$MP = P$$

$$Mx = Dx$$

$$My = \overline{P} \cdot Dy$$

Un'ultimo esempio che analizzeremo è quello del confronto di due numeri A e

B, ciascuno di due bit. L'output Y del circuito da realizzare dovrà essere alto se $A > B$, basso nel caso opposto. La tabella delle verità che riassume le situazioni in cui è $A > B$, è la 1.23.

A		B		Y
0	1	0	0	1
1	0	0	0	1
1	1	0	0	1
1	0	0	1	1
1	1	0	1	1
1	1	1	0	1

Tabella 1.23: Tabella delle verità per i casi in cui il numero A (a due bit) è maggiore del numero B (a due bit).

La corrispondente mappa K è mostrata in figura 1.23.

Figura 1.23: Mappa di Karnaugh relativa alla tabella delle verità 1.23.

Da questa, come è facile verificare, si ottiene:

$$Y = \overline{B}_1 \cdot A_1 + A_0 \cdot \overline{B}_0 \cdot \overline{B}_1 + A_0 \cdot A_1 \cdot \overline{B}_0$$

Il circuito che implementa tale funzione è mostrato in figura 1.24.

Figura 1.24: Circuito relativo alla tabella di verità 1.23.

1.8 Operazioni aritmetiche

Vediamo ora come, facendo uso di circuiti elementari che implementano funzioni logiche, sia possibile effettuare la somma o la differenza di numeri binari. Immaginiamo di voler effettuare la somma di due bit a e b. Indichiamo con Σ la somma e con R il riporto. È facile verificare che essi sono dati dalla tabella delle verità mostrata nella 1.24.

Vediamo facilmente che Σ coincide con $a \oplus b$, mentre R coincide con $a \cdot b$.

Immaginiamo di avere due numeri, di n bit ciascuno e contiamo i bit di ciascuno dei numeri a partire dal bit meno significativo (il più a destra di tutti) da 0 ad (n-1). Cioè:

$$\underline{a} \;=\; a_{n-1}a_{n-2}\cdots a_3a_2a_1a_0$$

$$\underline{b} \;=\; b_{n-1}b_{n-2}\cdots b_3b_2b_1b_0$$

a	b	Σ	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabella 1.24: Tabella delle verità per la somma ed il riporto di due bit.

La somma del bit i^{emo} di $\underline{a}$ e del corrispondente di $\underline{b}$ potrà essere ottenuta tramite un circuito che calcoli l'OR esclusivo e l'AND di a_i e b_i. Un tale circuito va sotto il nome di "semisommatore" o "Half Adder" (HA). In termini circuitali avremo quanto mostrato in figura 1.25.

Figura 1.25: Circuito "semisommatore" o "Half Adder" (HA).

Per effettuare la somma dei numeri a e b occorrerà un circuito che sommi alla somma dei due bit a_i e b_i, il riporto R_{i-1} dalla somma dei due bit precedenti (meno significativi). Tale circuito può in linea di principio essere ottenuto da due H.A., nel modo indicato in figura 1.26.

Figura 1.26: Circuito che effettua la somma dei due bit a_i, b_i e del riporto dalla somma dei due bit precedenti.

E' facile verificare che il riporto finale R_i è l'OR del riporto R'_i dal primo semisommatore e di quello R''_i dal secondo.

Un modo più pratico di effettuare la somma dei due bit e del riporto della somma dei due bit precedenti è ottenibile analizzando la tabella delle verità per i termini a_i, b_i, R_{i-1}, Σ_i, R_i mostrata nella tabella 1.25.

Dalla tabella delle verità si ottiene:

$$\Sigma_i = \overline{a}_i b_i \overline{R}_{i-1} + a_i \overline{b}_i \overline{R}_{i-1} + \overline{a}_i \overline{b}_i R_{i-1} + a_i b_i R_{i-1}$$

$$R_i = a_i b_i \overline{R}_{i-1} + \overline{a}_i b_i R_{i-1} + a_i \overline{b}_i R_{i-1} + a_i b_i R_{i-1}$$

Notiamo che R_i può essere semplificato in:

$$R_i = b_i R_{i-1} + a_i b_i + a_i R_{i-1} = a_i b_i + R_{i-1} \cdot (a_i + b_i)$$

La somma Σ_i è esprimibile come:

$$R_{i-1} \cdot (a_i b_i + \overline{a}_i \overline{b}_i) + \overline{R}_{i-1} \cdot (a_i \overline{b}_i + \overline{a}_i b_i)$$

Il termine nella prima parentesi è: $\overline{a_i \oplus b_i}$, quello nella seconda è: $a_i \oplus b_i$. Ne segue:

$$\Sigma_i = R_{i-1} \cdot \overline{a_i \oplus b_i} + \overline{R}_{i-1} \cdot (a_i \oplus b_i)$$

a_i	b_i	R_{i-1}	Σ_i	R_i
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Tabella 1.25: Tabella delle verità per la somma di due bit e del riporto della somma dei due bit precedenti.

È evidente allora che questa altro non è che:

$$\Sigma_i = R_{i-1} \oplus (a_i \oplus b_i)$$

La somma ed il riporto sono allora entrambi ottenibili dal circuito di figura 1.27.

Figura 1.27: Circuito che effettua la somma dei due bit a_i, b_i e del riporto dalla somma dei due bit precedenti.

Vediamo in tal modo che Σ_i ed R_i possono facilmente essere realizzati utilizzando porte logiche elementari.

1.9 Sottrazione binaria

Ammettiamo di voler sottrarre un numero binario A da un numero binario B, con $B > A$. Possiamo illustrare il procedimento da seguire se fissiamo il numero di bit di cui A e B sono costituiti. Ammettiamo ad esempio di dover adoperare numeri di 4 bit. Notiamo intanto che la somma di A e del suo complemento è:

$$A + \overline{A} \;=\; 1111$$

ciò che di solito si esprime dicendo che $\overline{A}$ è il *complemento ad 1* di A.

Nel seguito di questa sezione, per evitare di confondere il simbolo di somma con quello (+) adoperato per la funzione OR, adopereremo la lettera greca Σ per indicare la somma e la lettera Δ per indicare la differenza.

Notiamo che la relazione:

$$A\Sigma\overline{A} \;=\; 1111 \tag{1.5}$$

implica:

$$A\Sigma\left(\overline{A}\Sigma 1\right) \;=\; 10000 \tag{1.6}$$

Si dice anche che $(\overline{A}\Sigma 1)$ è il *complemento a 2* di A. Dalla 1.6 segue che:

$$A \;=\; 10000\Delta\overline{A}\Delta 1$$

e quindi:

$$B\Delta A \;=\; \left(B\Sigma\overline{A}\Sigma 1\right)\;\Delta 10000 \tag{1.7}$$

Dalla 1.7 segue che per sottrarre A da B occorre:

a complementare A, ottenendo $\overline{A}$

b sommare B ad $\overline{A}$

c sommare 1 al risultato

d lasciar cadere il quinto bit (il più significativo)

Vediamo quindi che, con il metodo suggerito, riusciamo a realizzare l'operazione differenza effettuando operazioni che abbiamo già appreso, cioè il complemento (NOT) di ogni bit di un numero e la somma Σ. E' facile verificare che, con il procedimento indicato, se $B > A$, il quinto bit c'è sempre (cioè il numero ottenuto effettuando la somma: $B\Sigma\overline{A}\Sigma 1$ ha il quinto bit sempre uguale ad 1).

Possiamo ora verificare che, nel caso in cui il quinto bit sia 0, B è minore di A. Verificheremo anche che in tal caso la differenza (negativa) ha un valore assoluto pari al complemento di $(B\Sigma\overline{A})$.

Riscriviamo infatti la 1.7 nella forma:

$$B\Delta A \;=\; B\Sigma\overline{A}\Delta 1111 \;=\; \Delta\left[1111\Delta\left(B\Sigma\overline{A}\right)\right] \;=\; \Delta\overline{\left[B\Sigma\overline{A}\right]}$$

Poiché, se n è un numero binario di quattro bit, 1111-n è il complemento di n.

Vediamo un esempio, cominciando dal caso $B > A$:

$$B \;=\; 12 \;=\; 1100 \;\;;\;\; A \;=\; 9 \;=\; 1001$$

$$B\Delta A \;=\; B\Sigma\overline{A}\Sigma 1$$

dove: $\overline{A} = 0110$.

Esaminiamo la tabella 1.26.

B	1100
$\overline{A}$	0110
$B\Sigma\overline{A}$	10010
	1
$B\Sigma\overline{A}\Sigma 1$	10011

Tabella 1.26: Sottrazione binaria nel caso in cui $B > A$.

Lasciamo ora cadere il bit più significativo (il quinto) ed otteniamo:

$$B\Sigma\overline{A}\Sigma 1 - 10000 \;=\; 10011 \;-\; 10000 \;=\; 0011 \;=\; 3$$

Come abbiamo già notato, se B è maggiore di A, la somma di B e di $\overline{A}$ ha sempre il bit più significativo uguale ad 1. Ciò suggerisce di usare tale bit come addendo (cioè come l'1 da aggiungere nella terza delle operazioni sopra descritte (la (c)). Infatti, nel caso in cui B sia maggiore di A, il quinto bit varrà 1 e ciò è quel che occorre per realizzare la sequenza di operazioni sopra descritte. Se invece B è minore di A, la somma $B\Sigma\overline{A}$ avrà il bit più significativo uguale a 0 ed aggiungerlo quindi alla somma $B\Sigma\overline{A}$ non cambierà nulla.

Vediamolo nel caso concreto in cui:

$$B \;=\; 9 \;=\; 1001 \quad ; \quad A \;=\; 12 \;=\; 1100$$

Avremo la situazione mostrata nella tabella 1.27.

B	1001	
$\overline{A}$	0011	
$B\Sigma\overline{A}$	01100	il 5 bit è zero
	0	
$B\Sigma\overline{A}\Sigma 0$	01100	

Tabella 1.27: Sottrazione binaria nel caso in cui $A > B$.

Se complementiamo tale numero, troviamo:

$$\overline{B\Sigma\overline{A}} \;=\; 0011$$

da interpretare come numero negativo (cioè dobbiamo immaginare di aver aggiunto un segno meno).

Quanto detto ci suggerisce un metodo generale per effettuare la differenza di due numeri B ed A valido sempre (cioè per B maggiore, uguale o minore di A). Il metodo consiste nel:

a complementare A

b sommare B ad $\overline{A}$

c sommare al risultato il quinto bit (b5) della somma

d se b5=1, il risultato ottenuto al punto (c), dopo aver lasciato cadere il bit più significativo, rappresenta la differenza (positiva) $B\Delta A$

e se invece b5=0, il complemento del risultato ottenuto al punto (c) rappresenta il valore assoluto della differenza (negativa) $B\Delta A$.

Tutto ciò è riassunto schematicamente in figura 1.28.

Il circuito contenuto nella scatolina riceve in ingresso $\overline{A}$, B ed il quinto bit della somma di B ed $\overline{A}$. Se tale bit vale 0, la differenza $B\Delta A$ è negativa ed il suo valore assoluto è $\overline{S}$. Se invece c3=1, la differenza $B\Delta A$ (positiva) è data da S.

Figura 1.28: Circuito che effettua la sottrazione binaria tra A e B.

1.10 Rappresentazione dei numeri negativi

Come visto, per ottenere il complemento ad 1 di un numero binario, si complementano tutti i suoi bits. Una regola altrettanto semplice può essere adoperata per ottenere il complemento a 2: si esaminano i bit di cui il numero è costituito, a partire dalla destra. Si copiano i bit fino ad incontrare un bit uguale ad 1. A partire da quel punto si ricopia l'1 ed i complementi dei bit che seguono. Esempio:

$$C_2\,(100110100) \;=\; 01001100$$

Verifichiamo che $C_2(N) + N \;=\; 100000000$ (tabella 1.28).

10110100	+
01001100	=
100000000	

Tabella 1.28: Verifica della regola per la costruzione del complemento a 2 di un numero.

Così, se il primo bit sulla destra è un 1, il complemento a 2 può essere ottenuto inserendo alla sinistra di tale 1 i complementi di tutti i rimanenti bit.

E' ovvio da quanto detto che per rappresentare un numero negativo occorrerà aggiungere ai bit che rappresentano il numero, un ulteriore bit che ne rappresenti il segno. Le due convenzioni comunemente adoperate sono quella del "complemento a due" e quella del "complemento ad uno". Esaminiamo la prima di queste, ammettendo, per semplicità, di voler rappresentare numeri il cui valore assoluto sia al massimo il decimale 15 (1111 in binario). I numeri da 0 a 15 saranno i numeri binari di cinque bit, con il bit più significativo (il quinto, che ora indicherà il segno) uguale a 0 e con i quattro meno significativi che vanno da 0000 a 1111.

Il numero negativo corrispondente ad un dato numero binario positivo si costruisce facendo il complemento a 2 del numero dato. Così:

$$+11 \;=\; 01011$$

$$-11 \;=\; 10101$$

Notiamo che, in tale sistema, la differenza di due numeri si ottiene prendendo la somma del primo e del complemento a 2 del secondo in tutti i casi! Cioè sia nel caso in cui A sia maggiore di B che in quello in cui A è minore di B. Vediamo ciò con un esempio, cominciando dal caso in cui sia A che B sono positivi. La somma è ricavata come mostrato nella tabella 1.29.

Calcoliamo ora la differenza $A\Delta B$ (tabella 1.30).

Vediamo che ora il quinto bit è zero, mentre un 1 compare in sesta posizione. Ciò non crea alcun problema poiché nel caso specifico i nostri registri hanno solo 5 bit. La differenza è positiva ed uguale a 1. Calcoliamo ora $B\Delta A$ (tabella 1.31).

		binario	decimale	
A	=	00110	6	+
B	=	00101	5	=
$A\Sigma B$	=	01011	11	

Tabella 1.29: Somma dei due numeri A e B.

		binario	decimale	
A	=	00110	6	+
$-B = C_2(B)$	=	11011	-5	=
$A\Sigma C_2(B)$	=	100001	1	

Tabella 1.30: Differenza dei due numeri: $A\Delta B$.

		binario	decimale	
B	=	00101	5	+
$-A = C_2(A)$	=	11010	-6	=
$B\Sigma C_2(A)$	=	11111	-1	

Tabella 1.31: Differenza dei due numeri: $B\Delta A$.

Vediamo che il risultato è negativo (1 in quinta posizione) e che il modulo vale:

$$C_2(11111) \; = \; 00001 \; = \; 1$$

Un sistema analogo è quello della rappresentazione in "complemento ad 1".

Esamineremo più avanti l'implementazione delle altre operazioni aritmetiche elementari.

1.11 Comparatori digitali

Ammettiamo di voler comparare due semplici bit A e B. Le possibili configurazioni sono quelle mostrate nella tabella 1.32:

A	B	Test	Y
0	0	$A = B$	$\overline{A} \cdot \overline{B}$
0	1	$A < B$	$\overline{A} \cdot B$
1	0	$A > B$	$A \cdot \overline{B}$
1	1	$A = B$	$A \cdot B$

Tabella 1.32: Comparazione tra due bit.

La variabile Y nell'ultima colonna definisce in modo univoco lo stato di A e B. Notiamo che l'uguaglianza $A = B$, è espressa da:

$$Y \;=\; A \cdot B + \overline{A} \cdot \overline{B}$$

cioè, facendo uso delle leggi di De Morgan:

$$\begin{aligned} Y \;=\; \overline{\overline{Y}} \;=\; \overline{\overline{A \cdot B + \overline{A} \cdot \overline{B}}} \;=\; \overline{\overline{A \cdot B} \cdot \overline{\overline{A} \cdot \overline{B}}} = \\ = \overline{\left(\overline{A} + \overline{B}\right) \cdot (A + B)} \;=\; \overline{A \cdot \overline{B} + B \cdot \overline{A}} \end{aligned}$$

In definitiva abbiamo la situazione mostrata in tabella 1.33.

$A > B$	$Y \;=\; A \cdot \overline{B} \;=\; 1$
$A < B$	$Y \;=\; B \cdot \overline{A} \;=\; 1$
$A = B$	$E \;\equiv\; Y \;=\; \overline{A \cdot \overline{B} + B \cdot \overline{A}} \;=\; 1$

Tabella 1.33: Comparazione tra due bit.

Un circuito che fornisce in uscita la variabile Y è quello di figura 1.29.

Figura 1.29: Circuito che implementa la comparazione tra due bit.

Se vogliamo comparare dei numeri costituiti da più bit, il circuito dovrà comparare i singoli bit ed un po' di logica aggiuntiva sarà necessaria per ottenere la risposta. Ammettiamo ad esempio di voler comparare due numeri di 4 bit. L'uguaglianza dei due numeri richiederà ovviamente l'uguaglianza due a due dei bit corrispondenti. Così, indicando con A_3, A_2, A_1, A_0 i bit del primo numero e con B_3, B_2, B_1, B_0 quelli del secondo, l'uguaglianza richiederà:

$$A_3 = B_3;\;\; A_2 = B_2;\;\; A_1 = B_1;\;\; A_0 = B_0$$

La prima di queste uguaglianze è espressa da $E_3 = 1$, la seconda da $E_2 = 1$, etc.. Avremo quindi l'uguaglianza richiesta se è:

$$E_3 \cdot E_2 \cdot E_1 \cdot E_0 \;=\; 1$$

A sarà invece maggiore di B se:

a $A_3 > B_3$ oppure:

b $A_3 = B_3$ e $A_2 > B_2$ oppure:

c $A_3 = B_3$ e $A_2 = B_2$ e $A_1 > B_1$ oppure:

d $A_3 = B_3$ e $A_2 = B_2$ e $A_1 = B_1$ e $A_0 > B_0$

Cioè:

$$Y \;=\; A_3 \cdot \overline{B_3} + E_3 \cdot A_2 \cdot \overline{B_2} + E_3 \cdot E_2 \cdot A_1 \cdot \overline{B_1} + E_3 \cdot E_2 \cdot E_1 \cdot A_0 \cdot \overline{B_0}$$

La condizione $A < B$ si realizza scambiando nell'ultima espressione A con B.

1.12 Parità di un numero binario

Dato un numero binario, si definisce *parità* la parità della somma dei bit. Così ad esempio: 1101 ha parità dispari (numero dispari di 1); mentre 1001 ha parità pari.

L'uso della parità è utile per controllare se un numero binario è stato trasmesso correttamente da un elemento di circuito ad un altro elemento, o anche da un sito ad un sito distante. Infatti il rumore che è inevitabilmente presente sulle linee usate per la trasmissione può far sì che un bit cambi valore. Se ammettiamo che la probabilità che, per effetto del rumore, 2 bit in una medesima parola (e.g. di 8 bit) cambino valore, sia trascurabile, possiamo verificare l'eventuale presenza di errori aggiungendo alla parola un ulteriore bit che ci dica quale deve essere la parità del numero. Così, se i numeri hanno 8 bit ed il numero in questione ha parità dispari, potremo aggiungere un nono bit di valore 1 che renda la parità complessiva pari. Se il numero di 8 bit aveva parità pari, aggiungeremo un nono bit di valore 0, con che la parità complessiva rimane pari. In ricezione, adopereremo un circuito che controlli la parità e verifichi che sia sempre pari. Se essa risulterà dispari il sistema denuncerà l'errore al circuito da cui l'informazione proviene, il quale invierà nuovamente il dato. Ovviamente, è possibile adottare la convenzione di parità dispari, anziché pari come nell'esempio fatto.

Esaminiamo ora un semplice circuito che genera il bit di parità per una parola di 4 bit, ammettendo una parità pari.

Se avessimo solo due bit, la parità sarebbe quella mostrata nella tabella 1.34.

A	B	parità del numero	bit da aggiungere
0	0	pari	0
0	1	dispari	1
1	0	dispari	1
1	1	pari	0

Tabella 1.34: Bit di parità per una parola a due bit.

Vediamo che il "bit da aggiungere" altro non è che l'XOR di A e B.

Se il sistema ha quattro bit ($ABCD$), potremo definire la parità della coppia (AB), quella della coppia (CD) ed infine quella dei due bit di parità così ottenuti. Occorrerà cioè una cascata di XOR, come mostrato in figura 1.30.

Figura 1.30: Circuito che implementa la parità di una parola a quattro bit.

1.13 Codificatori/Decodificatori

Un codificatore è un circuito che converte in un codice l'informazione corrispondente al verificarsi di uno tra n possibili eventi. Ad esempio, un codificatore è l'insieme dei circuiti presenti in un personal computer, che generano un numero binario (codice)

quando si pigia un determinato tasto. Un decodificatore è viceversa un circuito che riceve in ingresso un codice (cioè un numero binario, codificato in modo opportuno) ed attiva una di n possibili linee (o eventi).

Esaminiamo come esempio di decodificatore un circuito per la conversione BCD-Decimale. BCD sta per "Binary Coded Decimal" ed è un modo per rappresentare (codificare) i numeri decimali. In tale sistema i numeri da 0 a 9 sono rappresentati da 4 bit (tabella 1.35).

$$\begin{aligned} 0 &= 0000 \\ 1 &= 0001 \\ 2 &= 0010 \\ 3 &= 0011 \\ 4 &= 0100 \\ 5 &= 0101 \\ 6 &= 0110 \\ 7 &= 0111 \\ 8 &= 1000 \\ 9 &= 1001 \end{aligned}$$

Tabella 1.35: Codifica per il sistema "Binary Coded Decimal".

Un decodificatore BCD-Decimale riceverà in ingresso i quattro bit ed attiverà una di dieci possibili linee di uscita, corrispondenti ai numeri decimali da 0 a 9. Notiamo per inciso che nel codice BCD un numero decimale di più cifre è rappresentato da tanti gruppi di quattro bit quante sono le cifre decimali che lo costituiscono. Ad esempio:

371 =	0011	0111	0001
	3	7	1

Tabella 1.36: Rappresentazione nel codice BCD di un numero decimale a più cifre.

Se indichiamo con (a_3, a_2, a_1, a_0) i bit che costituiscono il codice BCD (con a_0 il meno significativo) e con $(\overline{a}_3, \overline{a}_2, \overline{a}_1, \overline{a}_0)$ i loro complementi, avremo:

$$\overline{a}_3 \cdot \overline{a}_2 \cdot \overline{a}_1 \cdot \overline{a}_0 \Rightarrow 0\ \textit{decimale}$$

$$\overline{a}_3 \cdot \overline{a}_2 \cdot \overline{a}_1 \cdot a_0 \Rightarrow 1\ \textit{decimale}$$

$$\overline{a}_3 \cdot \overline{a}_2 \cdot a_1 \cdot \overline{a}_0 \Rightarrow 2\ \textit{decimale}$$

$$\ldots$$

$$a_3 \cdot a_2 \cdot a_1 \cdot a_0 \Rightarrow 0\ \textit{decimale}$$

Un circuito che realizza tale funzione logica è quello mostrato in figura 1.31.
Tale circuito è noto anche come *decodificatore da 4 a 10 linee*.

Figura 1.31: Circuito che implementa un *decodificatore da 4 a 10 linee*.

Esaminiamo ora la struttura di un codificatore. Ammettiamo di voler generare un carattere (codice) BCD in uscita pigiando uno di 10 tasti possibili. Siano i tasti $T_0 \cdots T_9$. La tabella delle verità che realizza la funzione di codifica è la 1.37.

T_9	T_8	T_7	T_6	T_5	T_4	T_3	T_2	T_1	T_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

Tabella 1.37: Tabella delle verità che realizza la funzione di codifica per generare un carattere (codice) BCD.

Un tasto pigiato corrisponde alla variabile T uguale ad 1. Il tasto non pigiato corrisponde alla variabile T uguale a 0. Dalla tabella vediamo subito la corrispondenza tra le variabili Y e le T:

$$Y_0 = T_1 + T_3 + T_5 + T_7 + T_9$$

$$Y_1 = T_2 + T_3 + T_6 + T_7$$

$$Y_2 = T_4 + T_5 + T_6 + T_7$$

$$Y_3 = T_8 + T_9$$

Un circuito che realizza tale codifica può esser ottenuto da un integrato a matrice di diodi, come mostrato in figura 1.32.

Figura 1.32: Circuito che implementa la codifica di tabella 1.37.

Se uno degli interruttori $T_0 - T_9$ è chiuso, la corrispondente linea orizzontale viene a trovarsi a 5 V. Con ciò i diodi ad essa collegati avranno l'anodo a 5 V ed il catodo collegato alle resistenze R, a loro volta collegate a massa. In tali condizioni, la tensione sul catodo sarà 5-0.6=4.4 V, cioè ancora al livello alto. L'uscita (O le

uscite) collegate alla linea verticale cui è connesso il catodo del diodo in questione sarà quindi anch'essa al livello logico 1. Ciò implementa la funzione voluta.

Se in tale circuito due o più tasti vengono premuti contemporaneamente, il risultato può non essere quello desiderato. Un "codificatore! con priorità" elimina tale problema. In un codificatore di questo tipo, se due o più tasti sono pigiati insieme, l'azione che ha luogo è quella che si avrebbe pigiando solo il tasto più alto nella sequenza $T_0 \cdots T_9$. Per ottenere tale funzione dobbiamo modificare la tabella delle verità come mostrato nella tabella 1.38.

T_9	T_8	T_7	T_6	T_5	T_4	T_3	T_2	T_1	T_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	x	0	0	0	1
0	0	0	0	0	0	0	1	x	x	0	0	1	0
0	0	0	0	0	0	1	x	x	x	0	0	1	1
0	0	0	0	0	1	x	x	x	x	0	1	0	0
0	0	0	0	1	x	x	x	x	x	0	1	0	1
0	0	0	1	x	x	x	x	x	x	0	1	1	0
0	0	1	x	x	x	x	x	x	x	0	1	1	1
0	1	x	x	x	x	x	x	x	x	1	0	0	0
1	x	x	x	x	x	x	x	x	x	1	0	0	1

Tabella 1.38: Tabella di codifica con "priorità".

Dove la x sta ad indicare che è irrilevante se quel tasto sia pigiato o no. Se ora si pigia il tasto T_2 ma per errore si pigia contemporaneamente il tasto T_3, allora $T_3 \neq 0$ ed anche se $T_2 = 1$, diverranno "alti" i bit Y_1 ed Y_0. Vediamo come le variabili $Y_3 \cdots Y_0$ possano essere espresse in funzione di $T_9 \cdots T_0$. Esaminiamo Y_0:

$$Y_0 = T_1\overline{T}_2\overline{T}_3\overline{T}_4\overline{T}_5\overline{T}_6\overline{T}_7\overline{T}_8\overline{T}_9 + T_3\overline{T}_4\overline{T}_5\overline{T}_6\overline{T}_7\overline{T}_8\overline{T}_9 + \\ + T_5\overline{T}_6\overline{T}_7\overline{T}_8\overline{T}_9 + T_7\overline{T}_8\overline{T}_9 + T_9$$

Questa si può semplificare in:

$$Y_0 = \overline{T}_9\overline{T}_8\left(T_7 + \overline{T}_7 B\right) + T_9$$

con:

$$B = \overline{T}_6\overline{T}_5\overline{T}_4\overline{T}_3\overline{T}_2T_1 + \overline{T}_6\overline{T}_5\overline{T}_4T_3 + \overline{T}_6T_5 = \overline{T}_6\left(T_5 + \overline{T}_5 C\right)$$

dove:

$$C = \overline{T}_4\overline{T}_2T_1 + \overline{T}_4T_3 = \overline{T}_4\left(\overline{T}_2T_1 + T_3\right)$$

Ed infine:

$$Y_0 = T_9 + \overline{T}_8\left(T_7 + \overline{T}_6\left(T_5 + D\right)\right)$$

con:

$$D = \overline{T}_4 \left(\overline{T}_1 T_1 + T_3 \right)$$

Un decodificatore svolge un ruolo opposto; esso attiva una tra N linee d'uscita secondo l'informazione che riceve da due o più linee d'ingresso.

1.14 Multiplexer/Demultiplexer

Ammettiamo di voler "incanalare" su di un'unica linea dati che provengono da una di N linee diverse. Si può utilizzare a tale scopo un multiplexer. Il multiplexer ha N ingressi "dati", un ingresso "indirizzo", che specifica quale degli ingressi dati si vuol selezionare, ed un'unica uscita, sulla quale viene convogliato il dato. Un esempio di multiplexer è quello mostrato in figura 1.33

Figura 1.33: Multiplexer con quattro ingressi.

Vediamo che se gli ingressi "indirizzo" A e B sono entrambi uguali a 0, $\overline{A} = 1$ e $\overline{B} = 1$ e la prima porta AND sarà abilitata [2].

Quindi il segnale presente sulla linea d'ingresso X_0 sarà convogliato in uscita non appena l'ingresso di abilitazione S (Strobe) sia alto. Se A=1 e B=0, sarà abilitata la seconda porta e sarà quindi convogliato in uscita il segnale presente sulla linea X_1 , e così via.

Un multiplexer può esser utilizzato per effettuare la conversione parallelo-seriale. Ammettiamo infatti che ad un circuito arrivino dei bit, costituenti una "parola", su n'=n+1 linee parallele e che esso debba poi convogliare tali informazioni su di un'unica linea, come mostrato in figura 1.34.

Figura 1.34: Multiplexer che effettua la conversione parallelo-seriale.

Si potrà usare un multiplexer che riceve sulle linee x_0, x_1,...,x_n i bit da convogliare in uscita; gli ingressi di controllo (o di indirizzo) c_0, $c_1, \cdots, c_N$, dovranno ricevere in successione temporale regolare i valori 000, 100,...,010,...,110,... etc.. (con c_0 il bit meno significativo). In tal modo il bit presente sulla linea x_0 sarà convogliato per primo in uscita, seguirà poi x_1, poi x_2 etc. Ovviamente il numero degli ingressi n'=n+1 dovrà esser legato al numero di ingressi di controllo N'=N+1 dalla relazione $n' = 2^{N'} - 1$. Così, per $n' = 4$ avremo $N' = 2$, per $n' = 8$, $N' = 3$ etc.

Un demultiplexer convoglia invece su una di n'=n+1 possibili linee di uscita i segnali che riceve su di una linea d'ingresso, facendo uso dell'informazione (indirizzo) che riceve su N'=N+1 linee di controllo. E' utile far presente che qualsiasi decodificatore può essere usato come demultiplexer utilizzando tutte le linee d'ingresso meno una per la funzione di selezione delle uscite.

[2] si dice che una porta è abilitata se essa è in grado di rispondere a variazioni dei segnali presenti sugli ingressi diversi da quello/quelli considerati di abilitazione. Ad esempio, se uno degli ingressi (considerato *di abilitazione*) di una porta AND a quattro ingressi è uguale a 0 l'uscita sarà 0 qualunque siano i valori presenti sugli altri ingressi. Si dice che in tal caso la porta è disabilitata

Inoltre possiamo far vedere come un multiplexer possa essere adoperato per implementare una funzione logica. Consideriamo il multiplexer di figura 1.35.

Figura 1.35: Multiplexer a quattro ingressi.

Gli ingressi sono x_0, x_1, x_2, x_3. A e B sono gli ingressi di controllo (con A il meno significativo). S è lo Strobe. La tabella delle verità è quella di tabella 1.39.

B	A	S	Y
0	0	1	x_0
0	1	1	x_1
1	0	1	x_2
1	1	1	x_3

Tabella 1.39: Tabella delle verità per il multiplexer di figura 1.35.

Y può anche essere scritto come:

$$Y \; = \; x_0\overline{A}\,\overline{B} \; + x_1 A\overline{B} \; + x_2\overline{A}\,B + x_3 AB \tag{1.8}$$

Se ora volessimo implementare la seguente funzione logica:

$$Y \; = \; C\overline{B}\,\overline{A} \; + \overline{C}\,\overline{B}\,\overline{A} \; + C\overline{B}\,A + \overline{C}\,BA$$

notando che i primi due termini si possono semplificare in $\overline{B}\,\overline{A}$, cioè:

$$Y \; = \; \overline{B}\,\overline{A} \; + C\overline{B}\,A + \overline{C}\,BA$$

possiamo confrontare questa espressione con la 1.8, imponendo l'uguaglianza dei coefficienti che moltiplicano i termini $\overline{B}\,\overline{A}$, $\overline{B}\,A$, BA ottenendo:

$$x_0 = 1; \quad x_1 = C; \quad x_2 = 0; \quad x_3 = \overline{C}$$

Basterà quindi utilizzare il multiplexer indicato, assegnando agli ingressi x_0, x_1, x_2, x_3 i valori trovati, per implementare la funzione logica indicata.

I multiplexer/demultiplexer svolgono un ruolo di fondamentale importanza nei processori e più in generale in tutti i sistemi digitali. Esistono in commercio numerosi integrati dedicati allo scopo. In figura 1.36 è illustrato il multiplexer 74LS151.

Figura 1.36: Multiplexer di tipo commerciale 74LS151.

Gli ingressi sono indicati con $I_0 - I_7$. L'indirizzo, cioè il numero binario che rappresenta l'ingresso da selezionare, è impostato sui piedini $S_0 - S_2$. L'uscita Z ed il suo complemento sono disponibili sui piedini 5/6. Il piedino indicato con E (accompagnato da un'inversione), che in condizioni normali va tenuto basso, è

l'ingresso di *Enable*. Se si vuole inibire il trasferimento occorrerà portare tale ingresso al livello alto.

L'integrato 74LS138, il cui schema logico è mostrato in figura 1.37 è utilizzabile come demultiplexer da 1 ad 8.

Figura 1.37: Demultiplexer di tipo commerciale 74LS138.

1.15 Registri e memorie: i Flip-Flop

I circuiti logici esaminati finora, costituiti da porte AND, NOT, NAND, etc., o da combinazioni più o meno complesse di queste, sono comunemente noti come "circuiti logici combinatori". Essi stabiliscono una corrispondenza univoca tra i segnali applicati agli ingressi e quelli presenti in uscita.

Esiste una varietà di circuiti che funzionano invece in modo *sequenziale*, vale a dire che lo stato logico che parte del circuito acquista ad un dato istante è funzione dello stato presente in altre parti del circuito negli istanti precedenti. Ciò consente, eventualmente in associazione a circuiti combinatori, di realizzare sequenze complesse di funzioni, che si verificano con una cadenza predeterminata da opportuni segnali di *clock*.

L'elemento che è alla base di tutti i circuiti sequenziali è il "Flip-Flop" (FF), il cui uso più ovvio è nella realizzazione di memorie per computer, dove esso può immagazzinare l'informazione elementare costituita dal *bit*.

Perché infatti un numero possa essere elaborato da un processore, esso deve essere preventivamente immagazzinato in un insieme di celle elementari o registri. Un tipico registro può contenere ad esempio un byte (8 bit), una parola di 16 bit o una parola di 32 bit etc.

Alla base di un generico registro è la cella elementare di memoria, spesso costituita da un *bistabile*, o Flip-Flop (FF). Un FF è un circuito con due ingressi (detti ingressi di Set e di Reset) e due uscite, convenzionalmente chiamate Q e $\overline{Q}$, una complementare dell'altra.

Un bistabile è anche noto come "binario" o "Latch". Esso può, come vedremo, esser realizzato a partire da porte logiche (ad esempio porte NAND) o acquistato in forma integrata. Più FF possono poi esser collegati insieme per realizzare registri, contatori, etc..

1.15.1 Flip-Flop di tipo RS

Esiste un'ampia varietà di FF. Quello cui abbiamo accennato è il flip-flop di tipo RS. Un FF RS può assumere due stati: nello stato di *Set*, le uscite sono: $Q = 1$, $\overline{Q} = 0$. Nello stato di *Reset* esse sono: $Q = 0, \overline{Q} = 1$. Vediamo così che un FF può essere adoperato per memorizzare un bit. I valori delle uscite sono determinati dai valori dei segnali presenti sugli ingressi R, S, secondo la tabella delle verità mostrata nella 1.40.

Notiamo che lo stato in cui entrambi gli ingressi sono nello stato logico basso (R=0, S=0) è proibito. Lo stato in cui entrambi gli ingressi sono nello stato logico

R	S	Q_n	$\overline{Q}_n$
0	0	non consentito	
0	1	1	0
1	0	0	1
1	1	Q_{n-1}	$\overline{Q}_{n-1}$

Tabella 1.40: Tabella delle verità di un flip-flop di tipo RS.

alto (R=1, S=1) è uguale a quello precedente l'applicazione di tali segnali sugli ingressi ($Q_n = Q_{n-1}$).

Il FF di tipo RS considerato è di tipo "asincrono" dove si intende che la transizione da uno stato all'altro è determinata esclusivamente dall'applicazione di determinati segnali sugli ingressi R ed S ed indipendente da eventuali impulsi di clock presenti nel sistema. Per veder ciò più chiaramente, cominciamo con l'esaminare la struttura interna di un FF di tipo RS, che utilizza due porte NAND (figura 1.38).

Figura 1.38: Struttura interna di un FF di tipo RS.

Vediamo che se S=0 ed R=1, avremo $Q = 0$ e $\overline{Q} = 1$. Se S=1 ed R=0, si ha invece $Q = 1$ e $\overline{Q} = 0$. Se R ed S sono entrambi uguali ad 1, le uscite Q_n e $\overline{Q}_n$ dipendono soltanto da Q_{n-1} e $\overline{Q}_{n-1}$, cioè dai valori che Q e $\overline{Q}$ avevano prima dell'applicazione dei segnali su S ed R. In particolare, se era $Q_{n-1} = 0$ e $\overline{Q}_{n-1} = 1$, le uscite saranno $Q_n = 0$, $\overline{Q}_n = 1$. Se viceversa era $Q_{n-1} = 1$, $\overline{Q}_{n-1} = 0$, si avrà $Q_n = 1$, $\overline{Q}_n = 0$. Vediamo così che lo stato con R=S=1 non modifica le uscite Q e $\overline{Q}$.

Vediamo invece cosa accade nello stato "proibito" con R=S=0. Se un ingresso di una porta NAND è 0, l'uscita sarà 1 qualunque sia il segnale presente sull'altro ingresso (in tale situazione la porta è disabilitata). Vediamo quindi che, con R=S=0 avremmo $Q = \overline{Q} = 1$. Tale situazione è anomala, poiché vogliamo limitarci a due stati: $(Q = 1\,; \overline{Q} = 0)$, $(Q = 0\,; \overline{Q} = 1)$.

C'è inoltre l'ulteriore problema costituito dal fatto che, in una eventuale successiva transizione allo stato con S=R=1, lo stato delle uscite Q e $\overline{Q}$ verrebbe a dipendere dal ritardo relativo tra i segnali applicati agli ingressi S ed R. Se la transizione di S avviene prima di quella di R, il sistema andrà nello stato di Set ($Q = 1$), altrimenti andrà in quello di Reset ($\overline{Q} = 1$). Poiché il ritardo relativo è un elemento aleatorio è chiaro come tale situazione non sia accettabile. Vedremo in seguito come in alcuni tipi di FF (ad esempio i JK) non esista alcuno stato proibito.

Si può ottenere un circuito con una tabella delle verità sostanzialmente uguale a quell precedente, facendo precedere ciascuno dei due gates NAND da un ulteriore NAND e scambiando i ruoli di Q e $\overline{Q}$ come mostrato in figura 1.39:

Ammettiamo che gli ingressi di *Preset* e di *Clear* siano entrambi alti. Se l'*Enable* è alto, le porte NAND, in ingresso saranno "abilitate" (cioè le loro uscite, S' ed R', dipenderanno dai segnali presenti sugli ingressi S ed R). Ciò è mostrato nella

Figura 1.39: Struttura interna di un FF di tipo RS.

tabella 1.41, dove vediamo che ora lo stato "proibito" è quello che corrisponde ad R=S=1.

R	S	R'	S'	Q_n	$\overline{Q}_n$
0	0	1	1	Q_{n-1}	$\overline{Q}_{n-1}$
0	1	1	0	1	0
1	0	0	1	0	1
1	1	non consentito			

Tabella 1.41: Tabella delle verità di un FF di tipo RS.

Se l'ingresso di Enable è basso, le porte in ingresso risulteranno disabilitate (cioè non risponderanno più ai segnali presenti sugli ingressi R ed S). Avremo in tale situazione R'=S'=1 e $Q_n = Q_{n-1}$, $\overline{Q}_n = \overline{Q}_{n-1}$. Con gli ingressi di Preset e Clear entrambi alti, le due porte poste a valle risponderanno solo ai valori dei rimanenti ingressi.

L'ingresso che abbiamo chiamato "Enable" può essere utilizzato per un funzionamento "sincrono" di tale FF. Se cioè immaginiamo di aver applicato agli ingressi R ed S due valori (ad esempio R=1, S=0), ed applichiamo all'ingresso di Enable un impulso di clock, finché questo è basso le uscite Q e $\overline{Q}$ rimangono inalterate, mentre esse acquisteranno i valori $Q = 0$, $\overline{Q} = 1$ non appena il clock sia divenuto alto. L'impulso (positivo) del clock abilita cioè il trasferimento dei segnali presenti sugli ingressi R ed S, verso le uscite Q e $\overline{Q}$ rispettivamente.

Gli ingressi di Preset e Clear che, come si è detto, vanno normalmente tenuti alti, possono essere adoperati per fissare dei valori iniziali prestabiliti delle uscite. Immaginiamo infatti di tenere basso l'impulso di clock, con che S'=R'=1. Se ora poniamo Preset=1 e Clear=0 avremo $\overline{Q} = 1$, $Q = 0$. Se invece poniamo Preset=0 e Clear=1, avremo $\overline{Q} = 0$, $Q = 1$. Ponendo successivamente Preset=Clear=1 il FF potrà nuovamente rispondere ai livelli presenti sugli ingressi R, S.

1.15.2 Flip-Flop di tipo D

Se tra gli ingressi S ed R di un FF di tipo RS poniamo un invertitore (NOT) come mostrato in figura 1.40, otteniamo un FF di tipo D (DATA).

Figura 1.40: Struttura di un FF di tipo D.

La presenza dell'invertitore fa sì che se S=1, sia R=0 e viceversa. Tale FF ha un solo ingresso (D) al posto dei due ingressi separati S ed R. Il dato presente sull'ingresso D viene trasferito in Q quando l'impulso di clock diviene alto. Un

esempio di FF di tipo D è quello mostrato in figura 1.41. La configurazione dei terminali (*pins*) è mostrata in figura 1.42.

Figura 1.41: Specifiche di un FF di tipo D commerciale (74AHC74).

Figura 1.42: Configurazione dei "pins" e diagramma logico per un FF di tipo D commerciale (74AHC74).

Questo (74AHC74) è un chip che contiene 2 FF di tipo D. Ciascuno dei due FF contenuti nel chip ha, oltre all'ingresso D, gli ingressi di Preset e Clear, l'ingresso di clock e le uscite Q e $\overline{Q}$. [3]

Notiamo che in questo FF la transizione avviene nell'istante in cui il clock diviene alto. Una volta che tale livello del clock sia stato raggiunto, le uscite non risentono più di eventuali variazioni dell'ingresso D. Tale tipo di variazione dello stato di un FF è noto con il nome di "edge-triggering". Moltissimi dei FF disponibili in commercio sono di questo tipo. Gli ingressi asincroni ($\overline{S}_D$ e $\overline{R}_D$) sono *active low*, cioè hanno un effetto solo se vengono tenuti bassi. Durante il normale funzionamento del FF con trasferimento del dato (D) effettuato dal clock, essi vanno tenuti *alti*.

In un normale FF di tipo D, l'output segue l'input in corrispondenza al fronte di salita dell'impulso di clock. Esiste un tipo particolare di D FF, noto come Latch, in cui l'output diviene una copia dell'ingresso quando il clock è alto, e conserva tale valore quando il clock diviene basso. In altre parole, la differenza tra i due tipi è che un normale D FF è "edge-triggered" mentre un Latch è "level triggered". Data la similarità con il D, si conserva la dizione "registro di tipo D" per il normale D e si usa quella di "transparent Latch" per il nuovo registro (in cui l'output "si aggancia" all'ingresso per l'intera durata dell'impulso di clock).

Un esempio di FF di tipo D è il '574, mentre un Latch è il '573. Questi, come mostrato in figura 1.43 sono dei chip con 8 FF in ciascuna unità (sono cioè di tipo "octal"). Il '574 ha un ingresso di clock indicato con *cp* e con il simbolo $>$ che denota un trigger sul fronte di salita del clock. Il '573 ha l'analogo ingresso indicato con LE (Latch Enable). Entrambi hanno un ulteriore ingresso di abilitazione/disabilitazione, indicato con $OEBAR$.

Un tipo di latch adoperato nei microprocessori è il 74125, mostrato in figura 1.44. Questo è un buffer quadruplo "tri-state". Il singolo latch tri-state contenuto in

[3]Nella sigla che identifica il chip, le ultime due cifre (74) specificano la funzione effettuata (FF di tipo D nel nostro caso). Le prime due cifre numeriche (nuovamente 74, ma è un caso!) si riferiscono al range di temperatura in cui può operare: 74 implica un'applicazione "commerciale", per un range di temperatura che va da 0^oC a 70^oC. Se trovassimo un 54 al posto di tali due cifre, ciò implicherebbe un dispositivo adatto ad applicazioni "militari", in un range di temperature compreso tra -55^oC e $+125^oC$. Le due lettere intermedie (LS nel nostro caso) stanno poi ad indicare la classe o tipo del chip. LS sta per "Low power Schottky". Altre sigle sono: S=Schottky; AS=Advanced Schottky; HC=High-Speed CMOS; HCU=HCMOS unbuffered; HCT=HCMOS con inputs TTL. I tipi 74-HC, 74HCU, 74HCT lavorano nel range di temperature che va da -40^oC a $+85^oC$.

Figura 1.43: FF commerciali di tipo D, il '574, e Latch, il '573.

questo chip è mostrato in figura 1.45 insieme alla sua tabella delle verità. Vediamo che, se l'ingresso di controllo C è basso, l'uscita segue l'ingresso dati; se invece esso è alto, l'uscita è in uno stato di "alta impedenza". In tale stato il livello di tensione di un'eventuale linea cui tale uscita sia collegata, dipende solo dagli eventuali altri elementi cui la linea è collegata. In altre parole, nello stato di "alta impedenza" l'uscita del latch è sostanzialmente "isolata" (la sua impedenza verso massa è teoricamente infinita). Il nome dato a tale latch (tri-state) deriva appunto dal fatto che l'uscita può acquistare, oltre ai valori 0 ed 1, quello di "alta impedenza".

Figura 1.44: Latch tri-state 74LS125 adoperato nei microprocessori.

Figura 1.45: Tabella delle verità del latch tri-state 74LS125.

1.15.3 Flip-Flop di tipo JK

Un tipo di FF che ha un ampio campo di applicazioni è il JK. Questo differisce da quello di tipo RS nel fatto che ora lo stato in cui entrambi gli ingressi (che ora si chiamano J e K) sono alti è consentito. In tale stato, l'arrivo di un impulso di clock fa passare le uscite del FF negli stati complementari di quelli in cui erano prima: $Q_n = \overline{Q}_{n-1}$, $\overline{Q}_n = Q_{n-1}$. Ciò può essere ottenuto facendo precedere ciascuno dei due ingressi di un FF RS, da una porta AND, a sua volta pilotata dall'uscita complementare (la superiore da $\overline{Q}$, quella inferiore da Q), come mostrato in figura 1.46.

Cominciamo con l'esaminare il caso, particolarmente semplice, in cui sia J che K siano uguali a zero. È evidente dalla figura 1.46 che con J=0 è *disabilitata* la porta AND superiore, mentre con K=0 è *disabilitata* quella inferiore. Entrambe le porte, e quindi gli ingressi R ed S del FF, saranno uguali a zero. Con ciò lo stato delle uscite del FF non cambierà all'arrivo dell'impulso di clock.

Se J=1 e K=0, esaminiamo i due possibili stati di Q e $\overline{Q}$:

a) $Q = 1$, $\overline{Q} = 0$

In tal caso avremo S=0, R=0 e quindi lo stato delle uscite del FF non cambierà all'arrivo dell'impulso di clock.

b) $Q = 0$, $\overline{Q} = 1$

Ora avremo R=0, S=1. In tal caso, all'arrivo dell'impulso di clock le uscite acquisteranno i valori:

$Q = 1$, $\overline{Q} = 0$

Vediamo cioè che ponendo J=1 e K=0 noi *settiamo* il FF.

Figura 1.46: Struttura di un FF di tipo JK.

Se fosse stato J=0, K=1 possiamo anticipare, data la simmetria del circuito, che le uscite saranno $Q = 0$, $\overline{Q} = 1$ dopo l'applicazione di un impulso di clock. Verifichiamolo, esaminando i due possibili stati del FF:

a) $Q = 1$, $\overline{Q} = 0$

In tal caso avremo: S=0, R=1; da cui seguirà $Q = 0$, $\overline{Q} = 1$.

b) $Q = 0$, $\overline{Q} = 1$

In tal caso avremo: R=0, S=0 e quindi $Q_n = Q_{n-1}$.

Vediamo cioè che ponendo J=0 e K=1 noi *resettiamo* il FF.

Esaminiamo infine lo stato J=K=1. In tal caso, se era $Q_{n-1} = 1$, $\overline{Q}_{n-1} = 0$ avremo che la porta AND inferiore è abilitata (Q=1, K=1) mentre è disabilitata quella superiore (poiché $\overline{Q} = 0$). Ne segue che, con l'arrivo dell'impulso di clock sarà $Q = 0$, $\overline{Q} = 1$, cioè il sistema passerà nello stato complementare di quello in cui era. Il successivo impulso di clock troverà ora abilitata la porta superiore e disabilitata quella inferiore. Di conseguenza, con tale impulso di clock Q diventerà 1 e $\overline{Q}$ 0.

Il sistema cambia quindi stato ad ogni impulso di clock (fintantoché J e K vengono mantenuti fissi al livello alto). Ciò è mostrato nel grafico di figura 1.47:

Figura 1.47: Segnali di clock, ingresso ed uscita di un FF di tipo JK.

Notiamo che la frequenza dell'onda rettangolare $Q(\overline{Q})$ è la metà di quella del clock. Un FF di tipo JK con J=K=1 effettua quella che vien chiamata la funzione "TOGGLE" ed è adoperato come divisore di frequenza. Ponendone due o più in cascata si può realizzare una divisione della frequenza del clock per 4 o per potenze di 2 più elevate.

L'operazione di questo circuito potrà presentare dei problemi, poiché l'uscita (Q) tenderà ad oscillare tra 0 ed 1 nell'intervallo di tempo t_c in cui il clock è alto. Per esempio, se J=K=1 e Q=0, all'arrivo di un impulso di clock Q diverrà 1 dopo un piccolo ritardo Δt (dipendente dalla struttura interna del dispositivo). A questo punto avremo J=K=1, Ck=1 e Q=1. Ora Q tenderà a divenire 0, a meno che il Ck

non sia nel frattempo sceso a 0. L'uscita Q altrimenti oscillerà a lungo tra 0 ed 1 ed alla fine dell'impulso di clock il suo stato sarà casuale. Tale fenomeno va sotto il nome di "race-around-condition" ed ha luogo se $\Delta t \ll t_c$. Esso può essere eliminato facendo uso di:

- FF di tipo Master-Slave
- FF di tipo Edge-Triggered

Il principio su cui sono basati i primi è illustrato in figura 1.48 (dove ammetteremo che sia J=K=1).

Figura 1.48: Struttura di un FF JK di tipo Master-Slave.

Il primo dei due FF RS effettua la transizione quando il clock è alto. Tuttavia in tale intervallo di tempo lo stato di Q e $\overline{Q}$ (uscita del secondo FF) non può cambiare. Infatti, la presenza dell'invertitore prima del Ck_2 fa sì che in tale intervallo di tempo il secondo FF sia disabilitato. La transizione di Q_2 e $\overline{Q}_2$ avviene solo quando il segnale di clock diviene basso. Il comportamento di questo FF differisce da quello di un normale JK, per il fatto che la transizione dell'uscita avviene sull'impulso negativo del clock.

I FF del tipo edge-triggered non rispondono al livello del clock, ma al fronte di discesa di questo. In questo caso gli ingressi non hanno effetto sulle uscite se non in corrispondenza della transizione negativa dell'impulso di clock.

La figura 1.49 mostra un esempio di forme d'onda per un FF JK del tipo Master-Slave.

Figura 1.49: Segnali di clock, ingresso ed uscita di un FF JK di tipo Master-Slave.

Come esempi di FF Master-Slave JK possiamo menzionare il modello SN74LS76 della Motorola. Questo chip ha due FF JK, come mostrato in figura 1.50. Ciascuno dei due FF ha, oltre agli ingressi J e K, due ingressi di Set e Reset asincroni, denotati con $\overline{S}$ ed $\overline{C}$ nella tabella di figura.

Figura 1.50: Specifiche tecniche del modello SN74LS76 di FF JK Master-Slave.

La tabella 1.42 mostra un elenco di alcuni dei più comuni tipi di FF disponibili in commercio. Per ciò che rigurda i simboli che vengono normalmente adoperati dai produttori nelle specifiche dei FF, dobbiamo ricordare che un FF con transizione sul fronte di salita del clock è comunemente specificato con un $>$ sull'ingresso del clock. Un FF con transizione sul fronte di discesa del clock è indicato con un'inversione "$\circ$" prima del medesimo simbolo. Vedasi ad esempio la figura 1.51. Quello a sinistra è un FF di tipo D con transizione sul fronte di salita del clock, quello a destra è un JK con transizione sul fronte di discesa.

Dispositivo	Funzione
70	JK con ingressi di Preset e Clear
71	RS con ingressi di Preset e Clear
72	JK con ingressi di Preset e Clear
73	Dual JK
74	Dual tipo D
76	Dual JK
105	JK con clock e con ingressi di Preset e Clear
175	Quad tipo D
273	Octal tipo D
276	Quad JK
375	Octal tipo D

Tabella 1.42: Flip-flop disponibili in commercio.

Figura 1.51: *Sinistra:* simbolo di un FF di tipo D con transizione sul fronte di salita del clock; *destra:* simbolo di un FF JK con transizione sul fronte di discesa.

1.16 Shift-Registers

Gli *shift-registers* (SR) (o registri a scorrimento), sono essenzialmente delle catene di FF, opportunamente collegati tra loro ed utilizzati per memorizzare una "stringa" di bits. Esiste una grande varietà di SR. Consideriamo come primo esempio uno SR seriale realizzato con FF di tipo D. Ricordiamo che in tale tipo di FF il dato presente sull'ingresso D viene trasferito in uscita (Q) all'arrivo di un impulso di clock (in genere sul fronte di salita di questo). Uno SR costituito da FF di tipo D è mostrato in figura 1.52.

Figura 1.52: *Shift-register* costituito da quattro FF di tipo D.

Ammettiamo che un 1 (livello alto) sia applicato all'ingresso (D_0) del primo FF, mentre gli ingressi di Preset e Clear di tutti i FF vengono mantenuti alti. L'applicazione di un segnale di clock trasferirà tale dato in $Q_0 = D_1$. Il successivo segnale di clock trasferirà il dato in $Q_1 = D_2$, il terzo lo trasferirà in $Q_2 = D_3$ ed infine il quarto in Q_3 . Alla fine (ammettendo che D_0 sia messo a zero dopo il primo impulso di clock) le uscite Q_0, Q_1, Q_2, Q_3 conterranno nell'ordine i bits (0,0,0,1). Se applicassimo in ingresso la sequenza di bit (1001) ed in fase con tale applicazione gli impulsi di clock, avremmo la situazione illustrata in figura 1.53.

Vediamo che, dopo l'applicazione del quarto impulso di clock, nei registri $Q_0, \cdots Q_3$ sono contenuti i bit 1001. Abbiamo così trasformato una distribuzione seriale di bits, applicati sequenzialmente in ingresso, in una parallela, sulle 4 linee d'uscita $Q_3 - Q_0$.

Figura 1.53: Segnali di clock, ingresso ed uscita per uno *shift-register* costituito da quattro FF di tipo D.

Notiamo che, se vogliamo che il numero rimanga nel registro dobbiamo, dopo il quarto impulso di clock, arrestare quest'ultimo. I dati possono alternativamente esser prelevati in forma seriale applicando altri quattro impulsi di clock. Questo tipo di registro, che ha un ingresso seriale ed un'uscita parallela, è chiamato "Serial-In-Parallel-Out" (o SIPO).

1.16.1 Universal Shift-Register

Il circuito che ora esamineremo può essere utilizzato indifferentemente per immettere dei dati in parallelo ed estrarli in modo seriale, o viceversa. Esso può anche essere adoperato per leggere i dati in serie ed estrarli in serie, o per leggerli in parallelo ed estrarli in parallelo. A tale scopo si fa uso di una linea di controllo per l'abilitazione dell'input in parallelo. Un esempio di tale circuito (SN74LS195A) è mostrato in figura 1.54. Esso è noto come "Universal Shift Register".

Figura 1.54: "Universal Shift Register" SN74LS195A.

Tale dispositivo è utilizzabile in un'ampia classe di applicazioni, che vanno da operazioni di "shift" al conteggio ed all'imagazzinamento dei dati. Esso, operando a velocità elevate (dell'ordine di 40 MHz), può effettuare trasferimento dei dati parallelo/seriale, seriale parallelo e shift seriale.

La tabella delle verità mostrata in figura 1.55 illustra le caratteristiche di funzionamento.

Figura 1.55: Tabella delle verità per lo "Universal Shift Register" di figura 1.54.

I dati sono immagazzinati nei quattro FF mostrati in figura. Se l'ingresso indicato con $\overline{PE}$ (Preset Enable) è alto, i dati possono essere caricati in modo seriale attraverso gli ingressi J e $\overline{K}$ che sono in genere uniti insieme. Se infatti $J = \overline{K} = 1$, ammettendo che sia $Q_0 = 0$ ($\overline{Q}_0 = 1$) la prima delle tre porte AND nel primo blocco a sinistra in figura avrà l'uscita alta e, a causa della doppia inversione presente all'uscita della porta OR sottostante ed all'ingresso Set del primo FF, avremo $Set = 1$. Se Q_0 fosse stato uguale ad 1 ($\overline{Q}_0 = 0$), sarebbe stata abilitata la seconda anziché la prima delle due porte AND associate al primo blocco e l'uscita della porta OR sarebbe stata comunque uguale ad 1. Quindi, con $J = \overline{K} = 1$ l'ingresso di Set del primo FF sarà uguale all'ingresso J. Di conseguenza l'uscita Q_0 diverrà 1 con il successivo impulso di clock (più esattamente, sul fronte di salita di questo).

Notiamo per inciso che, con il $\overline{PE}$ alto, la terza porta AND associata al primo blocco è disabilitata. In tali condizioni l'uscita della porta sarà uguale a 0, qualunque sia il valore presente in P_0.

In modo analogo, se $J = \overline{K} = 0$, l'ingresso di Set del primo FF sarà uguale a 0 e quello di Reset 1. Con ciò, l'uscita Q_0 diverrà uguale a 0 in corrispondenza al fronte di salita del prossimo impulso di Clock. Tale caricamento seriale del dato è quello indicato nella seconda e terza riga della tabella (Shift-Set First Stage - Shift-Reset) di figura 1.55.

Esaminiamo ora ciò che accade con l'arrivo dei successivi impulsi di Clock. Notiamo che, con l'ingresso $\overline{PE}$ alto, la prima porta AND di ciascuno dei blocchi mostrati in figura è abilitata (e la terza nel primo blocco disabilitata). Se esaminiamo la prima porta AND associata al secondo blocco, vediamo che essa ha come ingresso Q_0. Il collegamento della porta OR sottostante è tale che il Set del relativo FF è anch'esso uguale a Q_0. Vediamo così che il successivo impulso di Clock farà si che Q_1 divenga uguale a Q_0 (che era 1 nel nostro esempio iniziale). In modo analogo vediamo che il collegamento dell'uscita Q_1 del secondo FF alla prima porta AND del terzo blocco farà si che l'uscita Q_2 divenga uguale a Q_0 con l'arrivo del terzo impulso di clock. Infine, il quarto impulso di Clock farà si che Q_3 divenga uguale a Q_0. Lo shift di quattro posizioni del dato impostato in J è così completato con l'applicazione di quattro impulsi di Clock.

Se il $\overline{PE}$ è basso, i dati possono essere caricati in parallelo facendo uso degli ingressi P_0, P_1, P_2, P_3. Infatti, con $\overline{PE} = 0$, delle porte AND presenti in ciascuno dei quattro blocchi, solo quelle all'estrema destra saranno abilitate. Ammettiamo ora ad esempio che sia: $P_0 = 1, P_1 = 0, P_2 = 1, P_3 = 0$. Ciò implicherà: $S_0 = 1, S_1 = 0, S_2 = 1, S_3 = 0$.

Questi dati saranno trasferiti in Q_0, Q_1, Q_2, Q_3 sul fronte di salita del successivo impulso di Clock. Si è in tal modo effettuato il *caricamento* dei dati impostati sugli ingressi $(P_0 \cdots P_3)$ in $(Q_0 \cdots Q_3)$. Ciò è indicato nell'ultima riga della tabella di figura 1.55.

Se ora si vuole effettuare lo shift dei dati verso destra, occorre mettere alto il $\overline{PE}$. In tal modo il successivo impulso di Clock troverà abilitata la prima porta AND di ciascun blocco, collegata all'uscita Q del blocco precedente. Il dato sarà così trasferito da Q_i a Q_{i+1}.

L'ingresso indicato con $\overline{MR}$ (Master Reset) può, come mostrato nella prima riga della medesima tabella, essere adoperato per porre uguali a zero le uscite $Q_0 \ldots Q_3$ di tutti i FF della catena.

Esaminiamo infine il funzionamento del circuito quando $(J, \overline{K})$ assumono i valori (0,1) o (1,0), con $\overline{PE}$ alto (quarta e quinta riga della tabella delle verità).

Se $J = 1$ e $\overline{K} = 0$, la prima delle porte AND presenti nel primo blocco di figura sarà abilitata, mentre la seconda è disabilitata. In tali condizioni l'uscita del primo AND dipenderà dal valore di $\overline{Q}_0$ e quindi di Q_0. Se $\overline{Q}_0$ è alto (Q_0 basso) l'uscita dell'AND sarà alta e tale risulterà quindi l'ingresso di Set del primo FF. Con ciò $Q_0 \to 1$ e $\overline{Q} \to 0$ con il successivo impulso di clock. Se fosse stato $Q_0 = 1$ la transizione causata dall'arrivo dell'impulso di Clock sarebbe stata opposta. Vediamo così che con $J = 1$ e $\overline{K} = 0$ il primo FF effettua la funzione "TOGGLE" caratteristica di un JK. Ciò è indicato come "Shift, Toggle First Stage" nella quarta riga della tabella.

Il secondo FF, come è facile verificare, acquisterà il valore di Q_0 (cioè 1) con il secondo impulso di Clock, il terzo FF con il terzo impulso di Clock etc.

Se invece è $J = 0$ e $\overline{K} = 1$, vediamo che verrà ad essere abilitata la seconda

delle porte AND presenti nel primo blocco. Poiché questa ha come terzo ingresso Q_0, vediamo che se $Q_0 = 0$ l'uscita di tale porta, e quindi l'ingresso di Set del FF, sarà bassa. Con ciò sarà alto l'ingresso di Reset del FF e Q_0 rimarrà uguale a 0. Se Q_0 fosse stato uguale ad 1 sarebbe invece stato alto l'ingresso di Set del FF e Q_0 sarebbe rimsasto uguale ad 1. Tale funzione è indicata con "Shift, Retain First Stage" nella quinta colonna della tabella.

Un registro a scorrimento simile per molti aspetti al precedente, ma con la possibilità di shift sia a destra che a sinistra, è il SN74LS194, la cui struttura è mostrata in figura 1.56 e la cui tabella delle verità è illustrata in figura 1.57.

Figura 1.56: "Universal Shift Register" SN74LS194A.

Figura 1.57: Tabella delle verità per lo "Universal Shift Register" di figura 1.56.

1.16.2 First-In, First-Out SR

Un ulteriore tipo di registro che è ampiamente adoperato è il cosiddetto "First-In, First-Out" (FIFO). Lo schema è mostrato in figura 1.58.

Figura 1.58: Shift Register di tipo "First-In, First-Out" (FIFO).

Come vediamo, esso consiste di un normale SR con possibilità di ingressi paralleli. Il dato in input è applicato alla porta AND G_D (in modo seriale). Le porte $G_0 - G_3$ sono abilitate dagli ingressi $Q_0 \cdots Q_3$, uno solo dei quali è normalmente alto. Un opportuno circuito di comando fa sì che gli ingressi Q_0, Q_1, Q_2, Q_3 siano abilitati consecutivamente in corrispondenza ai quattro impulsi di clock che agiscono su G_D abilitando l'immissione sulla linea L del dato corrispondente. Vediamo ciò più in dettaglio:

(a) il primo impulso di clock abilita G_D e manda $Q_0 \rightarrow 1$. Simultaneamente (o poco prima) il dato (primo bit) è immesso sulla linea L. L'abilitazione di G_0 fa sì che il dato sia caricato in Q_0.

(b) il secondo bit è immesso in ingresso ed il secondo impulso di clock applicato. Questo abilita G_D e manda $Q_0 \rightarrow 0$ e $Q_1 \rightarrow 1$. In tal modo G_1 viene ad essere abilitato ed il dato (secondo bit) caricato in Q_1 .

(c) il terzo bit è immesso in ingresso ed il terzo impulso di clock applicato. Questo abilita G_D e manda $Q_0, Q_1 \rightarrow 0$ e $Q_2 \rightarrow 1$. Ora G_2 sarà abilitato ed il terzo bit caricato in Q_2.

(d) il quarto bit è immesso in ingresso ed il quarto impulso di clock applicato. Questo abilita G_D e manda $Q_0, Q_1, Q_2 \rightarrow 0$ e $Q_3 \rightarrow 1$. Ora G_3 sarà abilitato ed il quarto bit caricato in Q_3.

Notiamo che i FF adoperati sono di tipo D, con ingressi di Preset e Clear. L'ingresso di Clear è quello indicato in figura mentre quello di Preset è ottenuto complementando questo mediante un invertitore (non indicato in figura).

La lettura è effettuata applicando impulsi di clock alla linea indicata da Ck_{out}. Ciascun impulso di clock genera uno shift a destra, di una posizione, di tutti i bit memorizzati. Da notare che il contatore (non indicato) che abilita in successione le porte AND $G_3 - G_0$ viene decrementato durante l'applicazione degli impulsi Ck_{out}. In altre parole ammettendo che dopo il quarto impulso di Ck_{in} avessimo $Q_0 = Q_1 = Q_2 = Q_3 = 0$, l'applicazione di un impulso di clock (Ck_{out}) pone $Q_0 = Q_1 = Q_2 = 0$ e $Q_3 = 1$. Se a questo punto immettiamo un nuovo dato, questo sarà caricato in FF3, che è il FF appena "liberato" con l'applicazione di Ck_{out}, mentre il dato presente all'uscita di FF3 viene trasferito all'uscita di FF2. Il dispositivo che effettua tale operazione è un "up-down" counter, che "conta" gli impulsi di clock che gli arrivano e mette degli opportuni 1 su quattro linee d'uscita, in sequenza crescente (cioè $0000, 0001, 0010, 0100, \cdots$) se il clock in arrivo è Ck_{in}; decrescente (cioè, $\cdots 1000, 0100, 0010, 0001, 0000$) se il clock in ingresso è Ck_{out}. Ciò è mostrato schematicamente in figura 1.59.

Figura 1.59: "Up-down" counter.

1.17 Applicazioni degli Shift-Registers e dei Flip-Flop

1.17.1 Applicazioni degli shift-registers

Come accennato, questi possono essere adoperati per memorizzare stringhe di bits, quali ad esempio bytes o parole di 16/32 bits. Essi hanno tuttavia molte altre applicazioni, non meno importanti. Notiamo intanto che se facciamo scorrere verso destra (right-shift) il contenuto di un registro, perdiamo il bit meno significativo. Se questo era uguale a 0, il numero immagazzinato viene ad essere diviso per due come conseguenza di tale operazione. Se esso era invece 1, allora il numero originario meno 1 viene ad essere diviso per 2. Viceversa, uno spostamento a sinistra (left-shift) moltiplica per 2 il numero che era stato inizialmente caricato nel registro. Vediamo quindi che, con successivi shift-left/shift-right possiamo moltiplicare/dividere per potenze di 2 un dato numero. Ovviamente, nel corso di uno shift-right perdiamo i bit meno significativi, mentre perdiamo quelli più significativi nel corso di uno shift-left. Questo non è un problema se il registro è molto lungo.

Un'altra applicazione degli SR è la conversione serie/parallelo e viceversa. Se carichiamo un numero, ad esempio di 8 bit, in uno SR e lo facciamo in parallelo sugli 8 FF di cui il registro è costituito, possiamo estrarre poi gli 8 bit in serie,

applicando 8 impulsi di clock. Viceversa, possiamo caricare gli 8 bit in modo seriale e leggerli in parallelo sulle 8 linee collegate ai piedini Q_i degli 8 FF.

Gli SR possono inoltre essere convenientemente utilizzati per generare sequenze predeterminate di impulsi, dove ciascun impulso può poi esser utilizzato per dar inizio ad una serie di operazioni. Immaginiamo infatti di collegare l'uscita dell'ultimo FF di uno SR come quello mostrato in figura 1.60, con l'ingresso di Set del primo FF.

Figura 1.60: Shift-Register in cui l'uscita dell'ultimo FF è collegata al Set del primo FF per generare sequenze predeterminate di impulsi.

Se nei FF A, B, C, D abbiamo inizialmente caricato i bit (0,0,0,1) rispettivamente, l'applicazione di successivi impulsi di clock farà "viaggiare" l'1 da Q_D in Q_A, poi da Q_A in Q_B, etc. Vediamo così che dopo 4 impulsi di clock l'1 è tornato in Q_D. Le linee Q_A, Q_B, Q_C, Q_D, assumeranno consecutivamente i valori mostrati nella tabella 1.43, dopo il numero di impulsi di clock indicato.

impulsi di Ck	Q_A	Q_B	Q_C	Q_D
0	0	0	0	1
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Tabella 1.43: Valori delle quattro uscite dello SR di figura 1.60 dopo l'applicazione di vari impulsi di clock.

Con tale dispositivo possiamo anche abilitare, a comando (tramite gli impulsi di clock) una delle quattro linee A, B, C, D (ad esempio, far scoccare la scintilla in una delle candele del motore di un'automobile).

Ma è anche possibile, con tale sistema, generare una sequenza predeterminata di bits. Ad esempio, se nei quattro FF avessimo inizialmente memorizzato la sequenza 1001, dopo l'applicazione di successivi impulsi di clock avremmo le sequenze indicate nella tabella 1.44. Un sistema analogo è quello, che esamineremo in una delle prossime sezioni, che consente la generazione di *sequenze pseudocasuali di bits*, di lunghezza arbitraria.

Notiamo ancora che, nel primo degli esempi visti, Q_D vale 1 ogni 4 impulsi di clock. Un tale sistema può quindi essere anche adoperato come "divisore di frequenza" (per 4 nell'esempio considerato, ma si può ottenere una divisione per potenze più elevate di 2, con registri a molti bit). Un tale tipo di utilizzo, dove l'uscita dell'ultimo FF è collegata all'ingresso del primo, va sotto il nome di "generatore di sequenze" o anche "contatore ad anello" (ring-counter).

Se i FF di cui il registro è costituito sono di tipo RS o JK, è possibile collegare l'uscita dell'ultimo FF della catena con l'ingresso R (o K) del primo. Si ottiene così quello che è noto come "Johnson Counter" o "Twisted-Ring Counter". Si può

impulsi di Ck	Q_A	Q_B	Q_C	Q_D
0	1	0	0	1
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

Tabella 1.44: Sequenza predeterminata di bits che si può ottenere con lo SR di figura 1.60.

verificare che con tale sistema si può ottenere una divisione di frequenza per 2N, dove N è la lunghezza del registro (anziché per N come in un normale ring-counter).

1.17.2 Applicazioni dei FF

Contatori asincroni

I FF sono ampiamente usati per realizzare dei contatori. Consideriamo come primo esempio quello di figura 1.61 (noto come ripple counter) che utilizza quattro FF di tipo JK. Questi sono del tipo *edge-triggered*, con transizione sul fronte di discesa del segnale di clock (come evidenziato dal cerchietto-inversione sull'ingresso di clock).

Figura 1.61: *Ripple counter* realizzato con quattro FF JK.

Ammettiamo che tutti gli ingressi J e K, come pure quelli di Preset e Clear, siano fissi ad 1. In tal caso i quattro FF effettueranno la funzione TOGGLE, cioè cambieranno stato ad ogni impulso presente sul rispettivo ingresso di clock. Più precisamente, la transizione delle uscite avverrà sul fronte di discesa dell'impulso di clock. Notiamo ora che per il secondo FF fa da clock l'uscita del primo; per il terzo, l'uscita del secondo etc. Di conseguenza le forme d'onda sulle uscite $Q_0 - Q_3$ saranno quelle mostrate in figura 1.62.

Figura 1.62: Segnale di clock ed uscite per il contatore di figura 1.61.

Vediamo che la frequenza di Q_0 è la metà di quella del clock, quella di Q_1 un quarto etc. Inoltre notiamo che la sequenza dei bit (Q_3, Q_2, Q_1, Q_0) inizia con (0,0,0,0) poi (0,0,0,1), diviene poi (0,0,1,0), (0,0,1,1) etc, cioè il registro $Q_3 \cdots Q_0$ "conta" gli impulsi di clock in arrivo al primo FF. Ovviamente, dopo il 15^{mo} impulso di clock il contatore si "resetta", cioè le uscite saranno (0,0,0,0), per poi ricominciare a crescere. Se colleghiamo ciascuna delle uscite ad un LED potremo vedere il conteggio (binario) del numero di impulsi di clock ricevuto dal FF0.

Il contatore, con una piccola modifica, può contare "all'indietro". E' sufficiente a tale scopo che il clock di ciascun FF sia collegato al $\overline{Q}$ (anziché al Q) del FF che lo precede. Ciò è facile da verificare.

E' possibile realizzare anche una catena di FF che, con un opportuno comando di controllo, conti in entrambi i versi. E' sufficiente collegare l'uscita di un FF al clock del successivo nel modo illustrato in figura 1.63.

Figura 1.63: Parte di una catena di FF che può contare in entrambi i versi.

Se l'ingresso X è alto, la porta AND B è disabilitata, mentre è abilitata la A. Il contatore esegue il ciclo in ordine crescente. Se invece l'ingresso X è basso, sarà abilitata (notare l'invertitore) la porta B e disabilitata la A. Il contatore eseguirà il ciclo in ordine decrescente.

E' possibile modificare i collegamenti nel circuito di figura 1.61, in modo da realizzare un contatore che, anziché resettarsi ogni 16 impulsi, si resetti ogni 10. Si ottiene in tal modo quello che è noto come un "contatore decimale" o Binary Coded Decimal (BCD).

Figura 1.64: "Contatore decimale" o Binary Coded Decimal (BCD).

La modifica è illustrata in figura 1.64 dove, come nel caso precedente, tutti i terminali J e K sono posti al livello logico alto. La presenza della porta AND, con ingressi collegati a Q1 e Q3 e con uscita collegata al Clear, fa sì che il contatore si resetti non appena Q1=Q3=1. Come si può vedere dalla figura 1.62, ciò accade dopo il decimo impulso di clock.

1.18 Contatori sincroni

I contatori fin qui esaminati sono noti come "contatori asincroni". Ciò poiché la transizione del secondo FF della catena avviene soltanto dopo la transizione del primo (essendo l'ingresso di clock del secondo alimentato dall'uscita (Q o $\overline{Q}$) del primo, ed analogamente per i successivi). Ciò costituisce uno svantaggio, poiché il tempo di propagazione (propagation delay) di un FF è relativamente lungo ($\tau \approx 20 - 30\,ns$ nel caso del '74). Se tutti i FF sono nello stato logico 1, tale tempo è massimo, poiché il successivo impulso di clock manda il primo FF da 1 a 0, questo a sua volta manda il secondo da 1 a 0 (con un ritardo τ), il secondo manda il terzo da 1 a 0 (con uguale ritardo) e così via. Il ritardo complessivo sarà in tal caso $N\tau$, se N è il numero di FF della catena. Per ridurre il tempo di propagazione conviene adottare uno schema in cui tutti i FF effettuano la transizione simultaneamente. Tale tipo di contatore va sotto il nome di "contatore sincrono". Un esempio è mostrato in figura 1.65.

Figura 1.65: Schema di un contatore sincrono.

Qui il primo FF (quello relativo alla cifra meno significativa) ha J=K=1, in modo da cambiare stato in corrispondenza ad ogni impulso di clock. Il secondo FF

(B) ha J=K=A, in modo da commutare (in corrispondenza all'arrivo di un segnale di clock) solo quando A=1. Il terzo FF (C) commuta quando sia A che B sono uguali ad 1. Infine D commuta se è verificata la condizione A=B=C=1. Come possiamo facilmente verificare (vedasi la tabella 1.45) la presenza delle porte AND soddisfa la logica insita nella successione dei numeri binari crescenti con (D, C, B, A) = (0,0,0,0), (0,0,0,1), (0,0,1,0), (0,0,1,1), ...

Clock	A	B	C	D
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
10	0	1	0	1
11	1	1	0	1
12	0	0	1	1
13	1	0	1	1
14	0	1	1	1
15	1	1	1	1
16	0	0	0	0

Tabella 1.45: Tabella dei valori assunti dalle uscite del contatore sincrono di figura 1.65 dopo ogni impulso di clock.

Infatti, se associamo il bit meno significativo al primo FF(A) e quello più significativo all'ultimo (D), la tabella ci dice che:

a il primo FF deve commutare ad ogni impulso di clock

b il secondo FF deve commutare in corrispondenza ad un impulso di clock se l'uscita del primo (A) vale 1

c il terzo FF deve commutare in corrispondenza ad un impulso di clock se sia l'uscita del primo (A) che quella del secondo (B) valgono 1

d l'ultimo FF deve commutare in corrispondenza ad un impulso di clock se tutte e tre le uscite (A, B, C) valgono 1

I collegamenti sono quindi quelli necessari per un conteggio crescente. Vediamo che, dopo il il quindicesimo impulso di clock il contatore si resetta automaticamente e ricomincia il conteggio.

Esaminiamo il tempo di propagazione in ciascuna delle transizioni indicate nella tabella, ignorando in un primo tempo, per semplicità, i ritardi dovuti alle porte AND. Nel passaggio dalla prima alla seconda riga il tempo di propagazione è solo quello (τ) relativo alla commutazione del primo FF (gli altri non cambiano stato). Analogamente, nella successiva transizione (passaggio dalla seconda all terza riga della tabella) il tempo di commutazione è solo quello relativo al secondo FF. Nella terza transizione cambiano stato i primi tre FF (A,B,C). Tuttavia il ritardo è ancora τ, poiché i tre cambiamenti di stato avvengono contemporaneamente. Proseguendo l'analisi è facile vedere che il ritardo complessivo nel conteggio fino a 15 è pari a $15\,\tau$. Questo risultato è da confrontare con quello che si incontra nel caso di un contatore asincrono, in cui il ritardo è $15\,\tau$ nella sola transizione (1 1 1 1 $\rightarrow$ 0 0 0 0).

In realtà, nel caso del contatore sincrono occorre tener conto del ritardo aggiuntivo (τ')dovuto alle porte AND, che è dell'ordine di (10-20 ns) per ciascun FF che alimenta una porta AND. Esaminiamo in dettaglio l'effetto del ritardo delle porte AND. Consideriamo ad esempio la transizione associata all'arrivo del quarto impulso di clock (quinta riga della tabella 1.45). Il passaggio dallo stato C=0 a C=1 può aver luogo solo dopo che:

(i) A sia diventato 1 (ritardo τ)

(ii) B sia diventato 1 (lo era già)

(iii) l'uscita della porta AND sia passata da 0 ad 1 (ritardo τ')

Vediamo così che la transizione di C da 0 ad 1 può *aver inizio* solo dopo un tempo $\tau + \tau'$. Se gli impulsi di clock si susseguono con un periodo $T < \tau + \tau'$, il sistema non potrà funzionare nel modo desiderato.

Un contatore sincrono binario a 4 bit, di uso abbastanza generale è il 74LS191. Esso può eseguire il conteggio in avanti o indietro a seconda del valore logico presente sull'ingresso di controllo "up/down". Il 74LS190, la cui struttura interna è mostrata in figura 1.66 e le cui caratteristiche sono mostrate in figura 1.67, è praticamente equivalente al 74LS191, con la sola differenza che esso è predisposto per il conteggio decimale (BCD).

Figura 1.66: Struttura interna del contatore sincrono binario a 4 bit 74LS190.

Figura 1.67: Specifiche tecniche del contatore sincrono binario a 4 bit 74LS190.

E' possibile, collegando due o più contatori in serie, realizzare conteggi fino a potenze più elevate di 2 (o di 10). Ad esempio, collegando in serie tre contatori BCD 74LS190 è possibile, come mostrato in figura 1.68, realizzare un contatore modulo 1000.

Vediamo che l'ingresso DN/UP (Down/Up) è collegato a massa, il che implica un conteggio crescente (massa=logico 0). L'ingresso di clock di tutti e tre i contatori

Figura 1.68: contatore modulo 1000 realizzato collegando in serie più contatori.

riceve il medesimo segnale d'ingresso (cioè gli impulsi da contare). Il 74LS190 è costruito in modo tale che sul piedino con la sigla RC (Ripple Clock Output) compaia un impulso "basso" (cioè uno 0) di durata uguale alla porzione "bassa" del clock d'ingresso, quando si verifichi un "overflow", cioè quando i quattro FF abbiano superato il massimo conteggio consentito (9 nel caso del 74LS190). Tale livello basso, collegato all'Enable (E) del contatore successivo, fa sì che questo cominci a contare. Così, quando il primo contatore (da sinistra) sia passato dal 9 allo 0, il secondo inizia a contare. Poiché però l'Enable ha la durata uguale a quella del livello "basso" del clock d'ingresso, esso conterà solo fino ad 1, mentre il primo contatore ripartirà da 0, arrivando fino a 9 e poi a 0. A questo punto il primo contatore genera un nuovo RC ed invia quindi un'Enable al secondo, che ora passerà a 2, e così di seguito. Quando anche il secondo contatore sia arrivato a 9 e poi a 0, esso genererà un RC e quindi invierà un'Enable al terzo, che inizierà a contare la centinaia.

1.19 Generazione di sequenze pseudocasuali

Un'applicazione interessante dei registri, che ha incontrato una larghissima diffusione, è quella che consente la generazione di sequenze di bits, anche molto lunghe, con distribuzione *casuale*. Per comprendere il concetto, pensiamo al problema statistico associato al lancio di una moneta. Il risultato del lancio può essere "testa" o "croce", con uguali probabilità.

La statistica ci insegna che in una serie di lanci successivi, la probabilità che il k^{mo} lancio dia un determinato risultato non dipende dai risultati ottenuti nei lanci precedenti. Se associamo al risultato "croce" il numero binario "0" ed al risultato "testa" il numero binario "1", in una successione di (k-1) lanci potremmo ad esempio aver trovato la seguente sequenza:

0 0 1 0 1 1 1 0 1 0 1 0 0 1 1 0

Questa è una sequenza che può esser considerata casuale. Il fatto di "conoscerla" non ci sarà di alcun aiuto nel tentare di prevedere il risultato del k^{mo} lancio.

Facendo uso di uno shift-register con clock, è possibile realizzare delle sequenze *approssimativamente* casuali. Si tratterebbe di sequenze realmente casuali se la lunghezza del registro fosse infinitamente lunga. Con un registro di lunghezza finita k, il numero di possibili combinazioni distinte di "1" e "0" à pari a 2^k (cioè al numero di *disposizioni* di due elementi su k posizioni). Ciò fa si che dopo un numero di bits pari a 2^k, la sequenza cominci a ripetersi in modo identico a se stessa; la sequenza ha una *periodicità*. Un'osservatore che abbia la pazienza di osservarla tutta sarà in grado di dire quali valori si succederanno a partire dal bit $2^k + 1$. Per un registro reale, diciamo di 33 bit, con un clock a 1 MHz, la sequenza inizia a ripetersi dopo circa 2 ore.

Vediamo con un esempio pratico come una tale sequenza possa esser realizzata nel caso di un registro costituito da quattro FF. La figura 1.69 illustra il circuito.

Figura 1.69: Generatore di sequenze pseudocasuali realizzato con quattro FF.

Come si vede, è sufficiente prendere l'XOR di due dei FF di cui il registro è costituito ed inviarlo in ingresso al primo. Possiamo facilmente verificare che, ammettendo di partire dalla configurazione 1 1 1 1 delle uscite, i valori successivamente acquistati dai quattro FF sono quelli mostrati nella tabella 1.46.

clock	FF_0	FF_1	FF_2	FF_3
-	1	1	1	1
↑	0	1	1	1
↑	0	0	1	1
↑	0	0	0	1
↑	1	0	0	0
↑	0	1	0	0
↑	0	0	1	0
↑	1	0	0	1
↑	1	1	0	0
↑	0	1	1	0
↑	1	0	1	1
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	0	1
↑	1	1	1	0
↑	1	1	1	1

Tabella 1.46: Sequenza degli stati per un generatore di sequenze pseudocasuali a quattro bit.

La sequenza dei valori assunti da uno qualsiasi dei FF, ad esempio da FF_0, è pseudocasuale, nel senso appena detto. La sequenza dei valori assunti da FF_1 differisce da quella di FF_0 solo per una traslazione temporale (circolare). Quindi, dal punto di vista statistico, essa è la medesima sequenza. Lo stesso dicasi per le sequenze dei valori assunti da FF_2 e FF_3.

Si può dimostrare che, per un registro di lunghezza 5, si ottiene una sequenza pseudocasuale di lunghezza $2^5 - 1 = 31$ facendo uso come ingressi dell'XOR delle uscite di FF_2 ed FF_4. Per registri di lunghezza maggiore, si dimostra che si ottengono le sequenze pseudocasuali di lunghezza massima $(2^k - 1)$ utilizzando XOR a 2 o più ingressi, come mostrato nella tabella 1.47 (dove i registri sono numerati a partire dallo 0).

È possibile generare sequenze pseudocasuali anche facendo uso, per gli ingressi degli XOR, di FF diversi da quelli indicati. Non in tutti i casi tuttavia le sequenze ottenute sono di lunghezza massima: esse possono avere delle sottosequenze che si

k	lunghezza della sequenza $2^k - 1$	ingressi all'XOR
5	31	2,4
6	63	4,5
7	127	5,6
8	255	1,2,6,7
9	511	4,8
10	1023	6,9
11	2047	8,10
12	4095	1,9,10,11
13	8191	0,10,11,12
14	16383	1,11,12,13
15	32767	13,14
16	65535	3,12,14,15

Tabella 1.47: Generatori di sequenze pseudocasuali. La tabella fornisce, per tutti i registri di lunghezza inferiore a 17, la massima lunghezza della sequenza e gli ingressi all'XOR. Si noti che tali ingressi sono numerati a partire dallo zero. Gli ingressi indicati non sono gli unici possibili: ad esempio, nel caso di un registro di lunghezza 7, una sequenza di lunghezza massima si ottiene anche adoperando come ingressi all'XOR le uscite 4 e 7.

ripetono all'interno della sequenza. Ulteriori dettagli sono discussi nell'appendice a questo capitolo.

Notiamo anche che, se tutti i FF di cui il registro è costituito assumessero il valore 0, lo stato successivo del sistema sarebbe sempre il medesimo. È per questo motivo che la lunghezza della sequenza è $2^k - 1$ e non 2^k: lo stato costituito di soli zeri non è presente. Ne segue che se in un ciclo completo contiamo il numero degli "1" e degli "0", il primo è sempre di una unità maggiore del secondo. Nel confronto con l'estrazione a sorte della monetina, ciò equivarrebbe a dire che le probabilità di "testa" o "croce" non sono esattamente uguali. La "testa" (0) è lievemente più probabile della "croce" (0). Ovviamente la differenza è tanto più piccola quanto maggiore la lunghezza della sequenza, e quindi la lunghezza del registro. Ad esempio, un registro a 23 bit darebbe 4194304 volte un risultato di "1" e 4194303 volte un risultato di "0".

I generatori di sequenze pseudocasuali hanno numerosissime applicazioni. Essi possono essere adoperati come generatori di rumore, ma un uso ancora più importante si ha nelle tecniche di protezione di dati da intrusioni. Una opportuna *chiave elettronica* fornita da un generatore di sequenze prefissato, e noto solo a chi invia ed a chi riceve il messaggio, può impedire ad altri di accedere all'informazione.

Una grandezza di notevole importanza nello studio delle sequenze pseudocasuali (ma più in generale nello studio di processi stocastici) è la *funzione di autocorrelazione*. Questa è definita come:

$$F(i) \;=\; \sum_{j=1}^{2^k-1} Q(j)Q(j+i) \quad (i = 0, 2^k - 1)$$

dove Q(j) è il valore di uno qualsiasi dei FF dopo il j^{mo} impulso di clock.

In altri termini, la funzione di autocorrelazione è la somma dei prodotti di ciascun bit della sequenza per il corrispondente della medesima sequenza, traslata (circolarmente) di una quantità i. I valori di Q(j) sono 0 o 1. È però pratica frequente, nel calcolare la funzione di autocorrelazione per sequenze pseudocasuali, sottrarre sistematicamente 0.5 e moltiplicare per 2, cioè:

$$Q(j) = 1 \rightarrow 1$$

$$Q(j) = 0 \rightarrow -1$$

Allora un generico prodotto varrà o -1 o +1. Per i=0 avremo che tutti i prodotti valgono 1, per cui:

$$F(0) = 2^k - 1$$

mentre si dimostra che, per $i \neq 0$ si trova: $F(i) = -1$. La F(i) è nuovamente uguale a $2^k - 1$ per $i = 2^k$, cioè se, nel calcolare i prodotti, trasliamo la sequenza di una quantità uguale al suo periodo.

La funzione di autocorrelazione può anche esser calcolata facendo uso della "forma d'onda" del segnale all'uscita di uno dei FF, ovvero dell'espressione analitica del segnale, anziché dalla sua rappresentazione "discreta". La funzione di autocorrelazione nel caso continuo è data da:

$$F(\tau) = \int_0^T Q(t)Q(t+\tau)d\tau$$

dove T è la durata temporale (periodo) della sequenza.

1.19.1 Generatori di sequenze casuali come sorgenti di rumore

Se l'uscita di uno dei FF del registro adoperato viene trasformata in un segnale analogico compreso tra -a e +a Volts, si ottiene un segnale la cui distribuzione in frequenza è approssimativamente piatta, almeno fino ad una certa frequenza. Più precisamente, la quantità che interessa è la *densità spettrale*, cioè la potenza media per unità di intervallo di frequenza.

Si dimostra nello studio dei processi di rumore che la densità spettrale così definita è legata alla funzione di autocorrelazione (nel caso continuo) attraverso il teorema di Wiener-Khintchine:

$$S_v(f) = 4\int_0^\infty Q(t)Q(t+\tau)cos(\omega\tau)d\tau$$

dove $S_v(f)$ è la densità spettrale, $Q(t)$ il segnale al tempo t e $Q(t+\tau)$ quello al tempo $t+\tau$. Nel nostro caso la $Q(t)$ è l'uscita del FF scelto, trasformata in modo da portarla nell'intervallo (-a,a). Si dimostra che il risultato è una serie di picchi molto stretti (funzioni δ) che hanno inizio alla frequenza f_{clock}/n e si ripetono ad intervalli di f_{clock}/n, dove f_{clock} è la frequenza del clock inviato al registro ed n la lunghezza della sequenza. Per n molto grande, i picchi saranno molto vicini ed i nostri strumenti di misura effettueranno di fatto una media su molti di questi

picchi in strette zone di frequenza. Vedremo cioè una curva praticamente continua, che approssima l'inviluppo della distribuzione "vera". Si trova che la forma di tale inviluppo è proporzionale a:

$$\frac{\sin^2(\pi f/f_{clock})}{(\pi f/f_{clock})^2}$$

Tale andamento è mostrato in figura 1.70.

Figura 1.70: Densità spettrale di un generatore di sequenze casuali.

La distribuzione è tutt'altro che piatta se consideriamo l'intero spettro di frequenze. Se però ci limitiamo a considerare la parte di bassa frequenza di questa distribuzione, diciamo $f < (0.1 \div 0.2)\ f_{clock}$, la distribuzione è praticamente piatta, come si può vedere nell'esempio di figura 1.71. Il rumore è praticamente *bianco*. Per ottenere tale risultato sarà quindi sufficiente far seguire al nostro generatore di rumore un opportuno filtro passa-basso.

Sono disponibili in commercio integrati appositi, basati su questi principi, che generano appunto del rumore bianco, nella zona di frequenza che si preferisce.

Figura 1.71: Densità spettrale di un generatore di sequenze casuali a basse frequenze: $f < (0.1 \div 0.2)\ f_{clock}$.

1.20 Alcune applicazioni pratiche di registri e contatori in esperimenti di fisica

Abbiamo già visto come una catena di FF possa essere utilizzata come divisore di frequenza. Un contatore può essere utilizzato, ovviamente, per contare. Ad esempio, se in un processo industriale si vuol contare il numero n di oggetti che sono passati su di un nastro trasportatore in un certo tempo sarà sufficiente far uso di un dispositivo (un diodo emettitore di luce (LED) ed una cellula fotoelettrica) per generare un impulso ogni volta che la luce emessa dal LED è interrotta dal passaggio di uno degli oggetti e non arriva quindi alla cellula fotoelettrica. Un contatore opportuno incrementerà di un'unità il conteggio ogni volta che gli arrivi un impulso.

Possiamo utilizzare un dispositivo simile per misurare l'intervallo di tempo tra due eventi. La figura 1.72 illustra un sistema di questo tipo.

Figura 1.72: Dispositivo per la misura dell'intervallo di tempo tra due eventi.

Gli "eventi" in questo caso sono costituiti dal passaggio di un oggetto attraverso due traguardi A e B (come nel caso del nastro trasportatore). Un primo impulso

viene generato al tempo t_A dal passaggio dell'oggetto davanti al traguardo A ed un secondo al tempo t_B in cui l'oggetto passa davanti al traguardo B.

L'impulso t_A sia inviato all'ingresso di Set di un FF di tipo RS (in cui era inizialmente $Q = 0$, $\overline{Q} = 1$) mentre l'impulso generato all'istante t_B sia inviato all'ingresso di Reset. Il gate AND che segue è collegato a valle ad un contatore (opportunamente calibrato), mentre dei due ingressi, uno è collegato al Q del FF e l'altro riceve la sequenza di impulsi di clock, di frequenza nota, fornita dal contatore.

Il funzionamento del circuito è il seguente: quando l'impulso generato al tempo t_A arriva al FF, questo avrà S=1, R=0 e quindi Q=1. Subito dopo sarà S=R=0 e quindi lo stato del FF rimane invariato fino al tempo t_B, quando R diventa 1 e quindi Q va a 0. Nell'intervallo di tempo $t_B - t_A$ in cui Q=1, la porta AND è abilitata e quindi essa lascia passare gli impulsi di clock in arrivo. Il contatore che segue conta tali impulsi di clock, il cui numero è proporzionale all'intervallo di tempo $t_B - t_A$.

Il sistema andrà ovviamente calibrato preventivamente. Un dubbio che può venire è il seguente. Sia l'impulso generato al tempo t_A che quello generato al tempo t_B sono prodotti dal medesimo dispositivo (cellula fotoelettrica seguita da monostabile[4]). Come possiamo far sì che dei due segnali generati su di una stessa linea, uno vada all'ingresso S e l'altro all'ingresso R? Ciò è semplice se sostituiamo il FF RS con un JK in cui J=K=1, come mostrato in figura 1.73.

Figura 1.73: Circuito che effettua la misura dell'intervallo di tempo tra due eventi nel caso in cui questi ultimi siano segnali prodotti dal medesimo dispositivo.

Se inizialmente era Q=0, l'arrivo del segnale generato al tempo t_A manda Q in 1, mentre il successivo arrivo del segnale generato al tempo t_B manda Q in 0 e $\overline{Q}$ in 1.

Un dispositivo molto simile a questo può essere adoperato se si vuole misurare una frequenza, cioè il numero di impulsi/sec. Tale dispositivo è mostrato in figura 1.74.

Figura 1.74: Dispositivo che effettua una misura di frequenza.

Gli impulsi in arrivo (di forma generica, ad esempio sinusoidale) vengono trasformati in impulsi rettangolari ed inviati ad uno degli ingressi di una porta AND. L'altro ingresso della porta è collegato all'uscita Q di un FF JK con J=K=1, che riceve sull'ingresso di clock un segnale ottenuto demoltiplicando per un fattore 10^6 un segnale di 1 Mhz. In tale circostanza Q è uguale ad 1 per un tempo di 1s. La porta AND sarà quindi abilitata per 1 s ed il contatore conterà il numero di impulsi che in tale tempo gli arrivano, cioè la frequenza voluta. Naturalmente occorrerà aggiungere qualcosa al circuito per far sì che alla fine dell'intervallo di tempo (di 1 s), il sistema si arresti e consenta la lettura del contatore.

[4]Un "monostabile" è un circuito che genera un impulso rettangolare di altezza e durata prefissata, quando riceva in ingresso un impulso (di forma arbitraria) che superi una soglia predeterminata

Anche se non specificamente relativi ad applicazioni in esperimenti di fisica, incontrano ampio utilizzo i contatori decimali. Infatti, in un contatore BCD (Binary Coded Decimal) un numero è espresso come una sequenza di cifre decimali. Ad esempio, il numero 3702 è memorizzato come mostrato nella tabella 1.48.

0011	0111	0000	0010
3	7	0	1

Tabella 1.48: Memorizzazione del numero 3702 in Binary Coded Decimal.

L'integrato 74LS93, che consente il conteggio fino a 16, può essere azzerato, come si è visto, in corrispondenza del numero 9. E' però più conveniente usare un contatore predisposto per contare in base 10, come il 74LS90.

Un'alternativa spesso adoperata è quella di un contatore con uscita esadecimale (base 16). Come visto, in tale rappresentazione, un numero e.g. di 2 bytes (16 bit) è rappresentato da una stringa di 4 caratteri esadecimali. Ad esempio, il numero binario: 1010 1101 0110 0011 è rappresentato in esadecimale da: $AD63$.

1.20.1 Codici adoperati nei computer

I simboli non numerici usati nella scrittura (alfabeto, punteggiatura, parentesi,...) nonché i comandi standard provenienti dalla tastiera di un computer o inviati ad una stampante, sono espressi nel codice ASCII (American Standard Code for Information Interchange). Questo è un codice a 7 bit, che consente quindi 128 combinazioni diverse. I principali caratteri ed i codici corrispondenti sono mostrati nella tabella di figura 1.75.

Nella tabella di figura 1.75, la prima colonna contiene il numero in decimale, la seconda il corrispondente valore esadecimale, la terza il corrispondente simbolo.

Figura 1.75: Tabelle delle corrispondenze tra caratteri e codice ASCII.

Un'estensione successiva, ad 8 bit, del codice ASCII fù sviluppata dall'IBM e prende il nome di codice EBCDIC (Extended Binary-Coded Decimal Interchange Code). Questo non è molto adoperato, all'infuori del mondo dei "mainframes" IBM.

1.21 Decodificatori e Display

Per visualizzare un numero binario abbiamo varie alternative. La prima è quella, ovvia, di visualizzare ogni singolo bit mediante un LED. Se il LED è acceso il bit corrispondente vale 1, se esso è spento vale 0. Tale soluzione non è pratica per la visualizzazione di numeri con molti bit. Essa inoltre non offre una lettura immediata del numero, più comoda se fatta in decimale o in esadecimale.

Utilizzando una decodifica BCD, dove occorre decodificare soltanto i numeri da 0 a 9 (in binario da (0,0,0,0) a (1,0,0,1)) si può far uso del chip 74LS42, che è

un integrato a 16 pin. Il display può essere effettuato utilizzando dei "display a 7 segmenti", in cui a ciascun numero BCD (tra 0 e 9) corrisponde l'accensione degli opportuni segmenti luminosi (LED), come mostrato in figura 1.76.

Figura 1.76: Visualizzazione dei numeri decimali con un display a 7 segmenti.

L'identificazione dei segmenti è quella mostrata in corrispondenza dell'8, in basso nella figura. Il decodificatore BCD a sette segmenti 74LS48, mostrato in figura 1.77, è un decoder/driver adatto a comandare i segmenti di un LED. Esso contiene sia la circuiteria che fornisce la decodifica dei quattro bit d'ingresso nelle 7 linee d'uscita, sia quella di alimentazione dei singoli segmenti.

Figura 1.77: Specifiche del decoder/driver 74LS48 usato per decodificare i quattro bit in ingresso e pilotare un diplay a 7 segmenti.

L'indicatore può essere spento portando a 0 l'ingresso di blanking ($\overline{BI}$: pin 4).

La tabella delle verità di tale circuito è mostrata, insieme al display riassuntivo e ad altre caratteristiche, in figura 1.78

Figura 1.78: Tabella delle verità per l'integrato 74LS48.

La decodifica per gruppi di 4 dal binario all'esadecimale può essere effettuata utilizzando l'integrato 74LS154, le cui caratteristiche sono mostrate in figura 1.79. Questo ha 16 uscite, corrispondenti ciascuna ad uno dei 16 simboli da 0 a 15 (F esadecimale). Ciascuna delle 16 uscite può essere collegata ad un opportuno LED, che si accende quando l'uscita corrispondente assuma il livello basso.

Figura 1.79: Specifiche dell'integrato 74LS154 usato per la decodifica da binario ad esadecimale.

Un circuito che effettua sullo stesso chip sia la decodifica che il display esadecimale è il TIL311. Questo riceve sui piedini (3,2,13,12) il carattere esadecimale (con il piedino 3 corrispondente al bit meno significativo) ed effettua un display esadecimale del relativo carattere (cioè da 0 ad F). Questo display è quello adoperato in alcune delle esperienze di laboratorio. Le caratteristiche di tale integrato sono mostrate nelle figure 1.80 e 1.81.

Figura 1.80: Layout del TIL311 usato per la decodifica e display esadecimale.

La figura 1.82 mostra uno schema a blocchi del TIL311.

Figura 1.81: Specifiche tecniche del TIL311.

Figura 1.82: Schema a blocchi del TIL311.

Notiamo la presenza di un "Latch" in ingresso. Questo è essenzialmente un insieme di 4 FF di tipo D. Esso ha lo scopo di "mantenere" i dati presentati in ingresso, per il tempo necessario alla decodifica ed al successivo display e lettura. Più precisamente, finché il segnale presente sul piedino 5 è basso, le uscita del Latch seguono l'ingresso (piedini 3,2,13,12). Quando il segnale è alto, le uscite non cambiano più. Si noti la presenza di due terminali d'ingresso (4 e 10) per l'accensione del punto decimale a destra ed a sinistra rispettivamente. Questi LED sono accesi attraverso i diodi indicati in figura. L'ingresso di "blanking" (piedino 8) è utilizzato per "abbuiare" il display. Ciò é ottenuto ponendo tale ingresso al livello alto. Se si vuole *attivare* il display, tale ingresso va messo al livello basso. Applicando a tale ingresso una sequenza di impulsi si avrà una modulazione dell'intensità luminosa.

1.22 Moltiplicazione e divisione in binario

L'operazione di moltiplicazione in binario tra due bit equivale all'AND di essi (tabella 1.49).

0	×	0	=	0
1	×	0	=	0
0	×	1	=	0
1	×	1	=	1

Tabella 1.49: Tabella relativa alla moltiplicazione in binario tra due bit.

La moltiplicazione tra numeri di più bit può essere effettuata con una semplice estrapolazione dell'analogo processo nel sistema decimale. Mostriamo alcuni esempi (tabelle 1.50 e 1.51).

Vediamo che ciascun bit del secondo numero (a partire dalla destra) deve essere moltiplicato per i bit del primo. A tali "moltiplicazioni" potranno provvedere delle porte AND. Il risultato ottenuto dalla moltiplicazione del primo bit andrà sommato al risultato della moltiplicazione del secondo, shiftato a sinistra di una posizione; questo andrà sommato al risultato della moltiplicazione del terzo bit, shiftato a sinistra di due posizioni ... etc. Vediamo così che un "moltiplicatore" può esser ottenuto effettuando sequenzialmente degli AND, degli shift e delle somme. Il circuito mostrato in figura 1.83 illustra una possibile implementazione di un moltiplicatore di due numeri a 3 bit.

Per il risultato della moltiplicazione è ovviamente necessario un registro a 6 bit. Il circuito comprende uno SR a 5 bit per il moltiplicando, uno SR a 3 bit per il

			1	0	1	1	=	11	×
			0	1	0	1	=	5	
			1	0	1	1			
		0	0	0	0				
	1	0	1	1					
0	0	0	0						
	1	1	0	1	1	1	=	55	

Tabella 1.50: Esempio di moltiplicazione tra numeri a più bit.

			1	1	0	1	=	13	×
			1	0	0	1	=	9	
			1	1	0	1			
		0	0	0	0				
	0	0	0	0					
1	1	0	1						
1	1	1	0	1	0	1	=	117	

Tabella 1.51: Esempio di moltiplicazione tra numeri a più bit.

Figura 1.83: Circuito che effettua la moltiplicazione di due numeri a 3 bit.

moltiplicatore ed un accumulatore a 6 bit, oltre ad un sommatore a 6 bit. Inizialmente si azzera il registro accumulatore, si caricano i tre bit del moltiplicatore nel relativo registro (con Q_0 corrispondente al bit meno significativo) ed i tre bit del moltiplicando nei registri $Q_0 - Q_2$ corrispondenti. Le uscite delle cinque porte NAND contengono a questo punto i prodotti del bit meno significativo del moltiplicatore (Q_0) per i tre bit del moltiplicando. Più in particolare, la porta indicata con e fornisce il prodotto dei due bit meno significativi dei due numeri, la porta d fornisce il prodotto del bit Q_0 del moltiplicatore per il bit Q_1 del moltiplicando, la porta c fornisce il prodotto del bit Q_0 del moltiplicatore per il bit Q_2 del moltiplicando. Le porte a e b sono per il momento al livello logico 0. I prodotti parziali così ottenuti sono ora disponibili agli ingressi del sommatore, in B_0, B_1, B_2. Essi sono anche disponibili all'uscita del sommatore (S_0, S_1, S_2) essendo gli altri ingressi (A_0, A_1, A_2) per il momento nulli. Al primo impulso di clock, il prodotto parziale (S_0, S_1, S_2) viene trasferito nei tre FF all'estrema destra del registro accumulatore e contemporaneamente agli ingressi (A_0, A_1, A_2) del sommatore. Il medesimo impulso di clock agisce sul registro moltiplicatore e sul registro moltiplicando nel modo seguente:

1. I bit del registro moltiplicatore vengono spostati in alto di una posizione, con che Q_1 viene trasferito in FF0

2. i tre bit del moltiplicando vengono spostati a sinistra di una posizione.

Notiamo che ora il registro all'estrema destra del moltiplicando è a livello logico 0. E' facile vedere che l'uscita della porta AND indicata con e è a livello logico 0, mentre le uscite dalle porte (d, c, b) contengono i prodotti del secondo bit del moltiplicatore per i tre bit del moltiplicando. Notiamo anche che con tale metodo, i tre bit ottenuti sono presenti agli ingressi (B_1, B_2, B_3) del sommatore e quindi risultano già shiftati a sinistra di una posizione, che è proprio quel che si fa nella moltiplicazione binaria. Ora all'uscita del sommatore comparirà, in $(S_0, S_1, S_2, S_3, S_4)$ la somma del risultato della prima moltiplicazione e di quello della seconda (shiftata). Il successivo impulso di clock trasferisce tali uscite nei corrispondenti FF dell'accumulatore e quindi agli ingressi $(A_0, ..., A_4)$ del sommatore.

Il medesimo impulso di clock causa uno shift del bit più significativo del registro moltiplicatore (che era a questo punto in FF1) nel registro FF0 e simultaneamente sposta di una posizione a sinistra i bit del registro moltiplicando. Notiamo che ora i registri FF0 ed FF1 del moltiplicando saranno a livello logico 0. Le uscite delle porte AND (c, b, a) forniscono ora i prodotti del bit più significativo del moltiplicatore per i tre bit del moltiplicando. Tali prodotti sono ora disponibili agli ingressi (B_2, B_3, B_4) del sommatore. Questo fornirà ora alle uscite $(S_0 - S_5)$ la somma finale richiesta. Sarà sufficiente un ulteriore impulso di clock per trasferirla nell'accumulatore e simultaneamente azzerare il registro moltiplicatore.

Il processo, effettuato come si è visto in modo seriale, è intrinsecamente lento. Esistono però in commercio dei moltiplicatori che effettuano le operazioni in parallelo, con un notevole guadagno in velocità.

1.23 La divisione in binario

L'idea base di un circuito digitale che effettua l'operazione di divisione tra due numeri interi (positivi) può essere illustrata con un semplice esempio, relativo al caso di numeri decimali. Ammettiamo infatti di voler dividere 33 per 7. Possiamo sottrarre il 7 dal 33 e verificare che la differenza (26) sia maggiore di 7. In tal caso sottraiamo da tale differenza ancora una volta il 7, ottenendo:

$$26 - 7 = 19$$

Procedendo ancora troviamo:

$$19 - 7 = 12$$

$$12 - 7 = 5$$

Poiché l'ultima differenza trovata è minore di 7, dobbiamo arrestare il processo e potremo dire che la parte intera del numero che esprime il rapporto è pari al numero delle sottrazioni effettuato, cioè 4. Dobbiamo ora calcolare la parte frazionaria del rapporto. A tale scopo sottraiamo ora 0.7 dall'ultima differenza trovata (5), tante volte quante ne occorrono per ottenere un resto inferiore a 0.7 (tabella 1.52).

Poiché il numero di sottrazioni effettuate è 7, concludiamo che 7 è la prima cifra decimale. Il rapporto desiderato, con la precisione di una cifra decimale è quindi 4.7. Per ottenere la successiva cifra possiamo procedere in modo analogo: sottraiamo 0.07 da 0.1 tante volte quante ne occorrono per trovare un resto inferiore a 0.07.

5	-	0.7	=	4.3	(1)
4.3	-	0.7	=	3.6	(2)
3.6	-	0.7	=	2.9	(3)
2.9	-	0.7	=	2.2	(4)
2.2	-	0.7	=	1.5	(5)
1.5	-	0.7	=	0.8	(6)
0.8	-	0.7	=	0.1	(7)

Tabella 1.52: Ricerca della prima cifra decimale del quoziente tra i numeri 33 e 7.

Troviamo così che la successiva cifra decimale è un 1, ed il rapporto è, a questo livello di precisione, 4.71. Il processo può essere ripetuto fino al livello di precisione voluto.

Ci proponiamo ora di vedere come il conteggio necessario per ottenere *la parte intera* del rapporto voluto possa essere implementato in logica binaria. Ammettiamo, per esemplificare, che divisore N e dividendo D abbiano 4 bit, e che il dividendo sia maggiore del divisore. Ad esempio sia D=14_D = 1110_B ed N=4_D = 0100_B. La prima sottrazione, effettuata con il metodo del complemento a 2, fornisce il risultato mostrato nella tabella 1.53.

		1	1	1	0	+
		1	1	0	0	=
carry →	1	1	0	1	0	

Tabella 1.53: Primo passo nella procedura di divisione tra i numeri 1110_B e 0100_B: sottrazione del secondo dal primo col metodo del complemento a 2.

Abbiamo cioè un bit di riporto (o di overflow), il quale indica che la differenza (uguale a $1010_B = 10_D$) è positiva. Sottraendo ancora abbiamo il risultato mostrato nella tabella 1.54.

		1	0	1	0	+
		1	1	0	0	=
carry →	1	0	1	1	0	

Tabella 1.54: Seconda sottrazione nella procedura di divisione.

Abbiamo ancora una differenza positiva (come evidenziato dalla presenza del bit di riporto) uguale a 6_D. Un'ulteriore sottrazione fornisce quanto si può vedere nella tabella 1.55.

cioè un numero positivo (2_D). Successivamente si ottiene quanto mostrato nella tabella 1.56.

Vediamo che ora il bit di riporto è 0, cioè la differenza è negativa. La parte intera del rapporto desiderato è pari al numero di volte che la differenza ha fornito un bit

		0	1	1	0	+
		1	1	0	0	=
$carry \rightarrow$	1	0	0	1	0	

Tabella 1.55: Terza sottrazione nella procedura di divisione.

		0	0	1	0	+
		1	1	0	0	=
$carry \rightarrow$	0	1	1	1	0	

Tabella 1.56: Quarta sottrazione nella procedura di divisione.

di riporto uguale ad 1, cioè 3_D. Un circuito che realizza la funzione desiderata è quello mostrato in figura 1.84.

Figura 1.84: Circuito che ralizza la divisione tra due numeri binari a 4 bit e fornisce in uscita la parte intera del rapporto.

Il dividendo è caricato in un registro, costituito da quattro FF di tipo D. Il sommatore indicato effettua la somma del contenuto di questo registro e del complemento a 2 del divisore, indicato in figura con $\hat{N}_3, \hat{N}_2, \hat{N}_1, \hat{N}_0$. Il dividendo è caricato inizialmente nel registro. Dopo che la prima somma sarà stata effettuata, il riporto $C_3 = 1$, mentre i registri Q_3, Q_2, Q_1, Q_0 conterranno il risultato della differenza tra dividendo e divisore. Poiché $C_3 = 1$, il prossimo impulso di clock farà avanzare il contatore di una unità. Ora il sommatore effettuerà la differenza tra il nuovo contenuto del registro ed il divisore. Se C_3 è ancora uguale ad 1, la porta AND continuerà ad essere abilitata ed il contatore avanzerà di una unità al prossimo impulso di clock. Il processo si ripete fino a quando C_3 non sia divenuto uguale a 0, al qual punto il contatore si arresta.

1.24 Appendice: Sequenze pseudocasuali

Abbiamo visto come un registro, con l'aggiunta di un feedback, possa esser adoperato per generate sequenze pseudocasuali. Si parla in tal caso di un *Linear Feedback Shift Register* o LFSR, dove l'aggettivo "Linear" sta ad indicare che il feedback è lineare.

I LFSR trovano applicazioni che vanno dalla criptografia alla misura del rate di errori nelle trasmissioni, a certi sistemi di comunicazione senza fili etc..

Discuteremo in questa appendice due aspetti collegati con i LFSR: il modo in cui il feedback è introdotto quando si voglia generare una sequenza pseudocasuale di lunghezza massima (m-sequenza) e la scelta dei punti (taps) di feedback.

Per quel che concerne il primo punto, si distingue in genere tra quella che è definita l'implementazione "alla Fibonacci" e l'altra, nota come l'implementazione "alla Galois".

La prima di queste è indicata in figura 1.85.

Figura 1.85: Implementazione "alla Fibonacci" di un *Linear Feedback Shift Register*.

Questa consiste in un registro in cui la somma pesata delle uscite binarie dei singoli stadi è riportata in ingresso. La somma è effettuata in "modulo 2", con che si intende che:

$$0 + 0 = 0$$

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

$$1 + 1 = 0$$

I pesi possono valere solo 0 o 1. I pesi g_0 e g_m possono solo valere 1.

L'implementazione "alla Galois" è invece mostrata in figura 1.86.

Figura 1.86: Implementazione "alla Galois" di un *Linear Feedback Shift Register*.

Qui il contenuto del registro è modificato, ad ogni impulso di clock, da un valore pesato in binario (con pesi 0 o 1) dell'output. Un'analisi dettagliata mostra che l'ordine dei pesi nella configurazione di Galois è opposto a quello della configurazione di Fibonacci, come indicato nelle figure. Se si scelgono pesi identici nelle due configurazioni, le due implementazioni del LFSR forniranno due sequenze che saranno uguali, ma con *fasi* diverse.

Nella realizzazione di un LFSR, la somma in modulo 2 viene effettuata facendo uso di porte XOR. Nell'implementazione alla Fibonacci il registro farà uso di uscite parallele e di un ingresso seriale. In quella di Galois farà invece uso di ingressi paralleli e di un'uscita seriale.

Se indichiamo con m la lunghezza del registro (cioè il numero di FF di cui esso è costituito) esiste una forma abbreviata per indicare i FF connessi nella rete di feedback. Nell'implementazione alla Galois, tale forma è del tipo:

$$[f_1, f_2, f_3, \cdots, f_j]_g$$

dove $j+1$ è il numero totale di FF collegati nella rete di feedback e con g_0 sempre uguale ad 1. $f_1 = m$ è il FF di ordine più elevato presente nella rete di feedback ed f_j sono i rimanenti FF presenti in tale rete. La sottoscritta g sta ad indicare la struttura di Galois.

L'analoga forma, nella struttura di Fibonacci, si indica con:

$$[f_1, m - f_2, m - f_3, \cdots, m - f_j]_f$$

Anche in tale caso g_0 è sempre uguale ad 1 ed $f_1 = m$.

Esaminiamo come esempio un LFSR di dimensione $m = 8$, con collegamenti di feedback a g_8, g_6, g_5, g_4 e con g_0 sottinteso. Nella forma di Galois questo sarà indicato con:

$$[8, 6, 5, 4]_g$$

ed in quella di Fibonacci con:

$$[8, 8-6, 8-5, 8-4]_f = [8, 2, 3, 4]_f$$

Normalmente i FF coinvolti nella rete di feedback vengono scritti in ordine decrescente. Inoltre il primo (quello con l''8' nell'esempio) è l'output del registro e la numerazione degli altri è decrescente andando verso l'input seriale.

Nelle tabelle che specificano i collegamenti di feedback, non è generalmente necessario specificare se si tratti dell'implementazione di Fibonacci o di Galois. Ciò poiché un dato set di collegamenti funzionerà con una qualsiasi delle due forme. L'unica differenza sarà costituita dal fatto che l'ordine della sequenza ottenuta in una caso è l'inverso di quella ottenuta nell'altro.

Veniamo ora alla determinazione dei collegamenti di feedback necessari per ottenere una sequenza pseudocasuale di lunghezza massima. Come già visto, se il LFSR ha lunghezza m, la lunghezza massima vale $N = 2^m - 1$. Con riferimento all'implementazione di Galois di figura 1.86 un generico LFSR può essere associato ad un "polinomio generatore" del tipo:

$$G(X) = g_m X^m + g_{m-1} X^{m-1} + g_{m-2} X^{m-2} + \cdots + g_2 X^2 + g_1 X + g_0$$

dove i coefficienti g_i rappresentano i "pesi" di figura 1.86. Ricordiamo che $g_1 = 1$ se l'i^{ma} connessione è presente; $g_1 = 0$ se essa è assente. A questi polinomi si applicano le normali regole dell'algebra, con la sola variante che ora tutte le operazioni algebriche sono effettuate in "modulo 2", cioè:

(a) per la somma:

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

(b) per il prodotto:

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 1$$
$$1 \cdot 1 = 1$$

Consideriamo ad esempio il polinomio:

$$G(X) = X^3 + X^1 + 1 \tag{1.9}$$

dove: $g_3 = 1$, $g_2 = 0$, $g_1 = 1$, $g_0 = 1$. Questo rappresenta il LFSR mostrato in figura 1.87.

Figura 1.87: LFSR che implementa l'esempio descritto.

Con la notazione di sopra, questo è:

$$[3, 1]_g$$

Veniamo ora alla regola generale per individuare i polinomi generatori associati a sequenze di lunghezza massima. La regola dice che il polinomio deve:

- esser "primo" (cioè non fattorizzato)
- essere un fattore di $X^N + 1$, dove $N = 2^m - 1$

dove tutte le operazioni vanno effettuate in modulo 2.

Consideriamo nuovamente l'esempio del polinomio 1.9. Notiamo che m=3 e che $N = 2^3 - 1 = 7$. Si può dimostrare che il polinomio 1.9 è primo e che esso è un fattore di $X^7 + 1$. Infatti è facile verificare che, modulo 2, è:

$$X^7 + 1 = (X+!)(X^3 + X + 1)(X^3 + X^2 + 1)$$

Dei tre polinomi a fattore nel secondo membro di quest'equazione, il secondo fornisce appunto la sequenza:

$$[3, 1]_g$$

Vediamo che una sequenza alternativa è fornita dal terzo polinomio a secondo membro, che dà:

$$[3, 2]_g$$

Il primo dei polinomi a secondo membro: $(X + 1)$ non è di ordine 3 e quindi non è un generatore per il nostro LFSR.

Notiamo ancora che, dato un LFSR di una certa lunghezza, il numero di combinazioni diverse che forniscono sequenze di lunghezza massima è sempre pari. Dato infatti l'insieme:

$$[f_1, f_2, f_3, \cdots, f_j]_g$$

esiste l'insieme alterantivo:

$$[f_1, m - f_2, m - f_3, \cdots, m - f_j]_g$$

che fornisce una sequenza speculare a quella fornita dal primo.

Le sequenze ottenute dalle diverse combinazioni di feedback differiscono solo per una fase.

Nelle tabelle che seguono, diamo i valori delle combinazioni di feedback per tutti i registri di lunghezza inferiore o uguale a 12.

10 stadi, 2 taps: (1 combinazione)
[10, 7]

10 stadi, 4 taps: (10 combinazioni)
[10, 9, 8, 5] [10, 9, 7, 6] [10, 9, 7, 3] [10, 9, 6, 1] [10, 9, 5, 2] [10, 9, 4, 2] [10, 8, 7, 5] [10, 8, 7, 2] [10, 8, 5, 4] [10, 8, 4, 3]

10 stadi, 6 taps: (14 combinazioni)

[10, 9, 8, 7, 5, 4] [10, 9, 8, 7, 4, 1] [10, 9, 8, 7, 3, 2] [10, 9, 8, 6, 5, 1] [10, 9, 8, 6, 4, 3]
[10, 9, 8, 6, 4, 2] [10, 9, 8, 6, 3, 2] [10, 9, 8, 6, 2, 1] [10, 9, 8, 5, 4, 3] [10, 9, 8, 4, 3, 2]
[10, 9, 7, 6, 4, 1] [10, 9, 7, 5, 4, 2] [10, 9, 6, 5, 4, 3] [10, 8, 7, 6, 5, 2]

10 stadi, 8 taps: (5 combinazioni)
[10, 9, 8, 7, 6, 5, 4, 3] [10, 9, 8, 7, 6, 5, 4, 1]
[10, 9, 8, 7, 6, 4, 3, 1] [10, 9, 8, 6, 5, 4, 3, 2]
[10, 9, 7, 6, 5, 4, 3, 2]

11 stadi, 2 taps: (1 set)
[11, 9]

11 stadi, 4 taps: (22 combinazioni)
[11, 10, 9, 7] [11, 10, 9, 5] [11, 10, 9, 2] [11, 10, 8, 6] [11, 10, 8, 1]
[11, 10, 7, 3] [11, 10, 7, 2] [11, 10, 6, 5] [11, 10, 4, 3] [11, 10, 3, 2]
[11, 9, 8, 6] [11, 9, 8, 4] [11, 9, 8, 3] [11, 9, 7, 4] [11, 9, 7, 2]
[11, 9, 6, 5] [11, 9, 6, 3] [11, 9, 5, 3] [11, 8, 6, 4] [11, 8, 6, 3]
[11, 7, 6, 5] [11, 7, 6, 4]

11 stadi, 6 taps: (40 combinazioni)
[11, 10, 9, 8, 7, 4] [11, 10, 9, 8, 7, 1] [11, 10, 9, 8, 5, 4]
[11, 10, 9, 8, 4, 3] [11, 10, 9, 8, 3, 1] [11, 10, 9, 7, 5, 1]
[11, 10, 9, 7, 4, 1] [11, 10, 9, 6, 5, 4] [11, 10, 9, 6, 4, 2]
[11, 10, 9, 6, 3, 1] [11, 10, 9, 6, 2, 1] [11, 10, 9, 5, 4, 3]
[11, 10, 9, 5, 4, 1] [11, 10, 9, 5, 3, 1] [11, 10, 9, 4, 3, 2]
[11, 10, 8, 7, 6, 5] [11, 10, 8, 7, 6, 3] [11, 10, 8, 7, 5, 3]
[11, 10, 8, 7, 4, 1] [11, 10, 8, 6, 5, 4] [11, 10, 8, 6, 5, 1]
[11, 10, 8, 6, 4, 3] [11, 10, 8, 6, 4, 2] [11, 10, 8, 5, 3, 2]
[11, 10, 8, 4, 3, 2] [11, 10, 7, 6, 5, 3] [11, 10, 7, 6, 5, 1]
[11, 10, 7, 6, 4, 2] [11, 10, 7, 6, 4, 1] [11, 10, 7, 6, 3, 2]
[11, 10, 7, 4, 3, 2] [11, 9, 8, 7, 6, 3] [11, 9, 8, 7, 4, 2]
[11, 9, 8, 6, 5, 2] [11, 9, 8, 6, 4, 3] [11, 9, 7, 6, 5, 4]
[11, 9, 7, 6, 5, 3] [11, 9, 7, 6, 4, 2] [11, 9, 6, 5, 4, 3]
[11, 8, 7, 6, 4, 3]

11 stadi, 8 taps: (25 combinazioni)
[11, 10, 9, 8, 7, 6, 5, 3] [11, 10, 9, 8, 7, 6, 5, 2] [11, 10, 9, 8, 7, 6, 4, 1]
[11, 10, 9, 8, 7, 6, 3, 2] [11, 10, 9, 8, 7, 5, 4, 2] [11, 10, 9, 8, 7, 5, 3, 2]
[11, 10, 9, 8, 7, 5, 2, 1] [11, 10, 9, 8, 7, 4, 3, 1] [11, 10, 9, 8, 7, 4, 2, 1]
[11, 10, 9, 8, 6, 5, 4, 2] [11, 10, 9, 8, 6, 5, 3, 2] [11, 10, 9, 8, 6, 5, 3, 1]
[11, 10, 9, 8, 6, 4, 3, 2] [11, 10, 9, 8, 5, 4, 2, 1] [11, 10, 9, 7, 6, 5, 4, 3]
[11, 10, 9, 7, 6, 5, 4, 1] [11, 10, 9, 7, 6, 4, 3, 2] [11, 10, 9, 7, 5, 4, 3, 1]
[11, 10, 9, 6, 5, 4, 3, 2] [11, 10, 9, 6, 5, 4, 3, 1] [11, 10, 8, 7, 6, 5, 4, 2]
[11, 10, 8, 7, 6, 4, 3, 1] [11, 10, 8, 7, 5, 4, 3, 2] [11, 9, 8, 7, 6, 5, 3, 2]
[11, 9, 8, 7, 6, 4, 3, 2]

12 stadi, 4 taps: (9 combinazioni)

[12, 11, 10, 4] [12, 11, 10, 2] [12, 11, 8, 6] [12, 11, 7, 4] [12, 10, 9, 3]
[12, 10, 5, 4] [12, 9, 8, 5] [12, 9, 7, 6] [12, 8, 6, 5]

12 stadi, 6 taps: (43 combinazioni)
[12, 11, 10, 9, 8, 4] [12, 11, 10, 9, 6, 4] [12, 11, 10, 9, 6, 2]
[12, 11, 10, 9, 4, 3] [12, 11, 10, 9, 4, 2] [12, 11, 10, 8, 7, 3]
[12, 11, 10, 8, 7, 2] [12, 11, 10, 8, 6, 4] [12, 11, 10, 8, 6, 1]
[12, 11, 10, 8, 5, 1] [12, 11, 10, 8, 2, 1] [12, 11, 10, 7, 6, 3]
[12, 11, 10, 7, 6, 2] [12, 11, 10, 7, 5, 2] [12, 11, 10, 7, 2, 1]
[12, 11, 10, 6, 3, 2] [12, 11, 9, 8, 7, 6] [12, 11, 9, 8, 7, 4]
[12, 11, 9, 8, 4, 1] [12, 11, 9, 8, 3, 1] [12, 11, 9, 7, 6, 5]
[12, 11, 9, 7, 6, 4] [12, 11, 9, 7, 3, 1] [12, 11, 9, 6, 5, 3]
[12, 11, 9, 5, 4, 1] [12, 11, 8, 7, 6, 3] [12, 11, 8, 7, 5, 4]
[12, 11, 8, 6, 5, 2] [12, 11, 8, 5, 4, 2] [12, 11, 7, 5, 4, 3]
[12, 11, 6, 4, 3, 2] [12, 10, 9, 8, 7, 6] [12, 10, 9, 8, 7, 3]
[12, 10, 9, 8, 6, 2] [12, 10, 9, 8, 4, 3] [12, 10, 9, 8, 4, 2]
[12, 10, 9, 7, 5, 2] [12, 10, 9, 6, 4, 3] [12, 10, 9, 6, 4, 2]
[12, 10, 8, 7, 6, 2] [12, 9, 8, 7, 5, 3] [12, 9, 8, 7, 4, 3]
[12, 9, 8, 6, 5, 4]

12 stadi, 8 taps: (18 combinazioni)
[12, 11, 10, 9, 8, 7, 6, 3] [12, 11, 10, 9, 8, 7, 4, 3] [12, 11, 10, 9, 8, 6, 5, 4]
[12, 11, 10, 9, 8, 5, 2, 1] [12, 11, 10, 9, 8, 4, 3, 1] [12, 11, 10, 9, 7, 6, 5, 1]
[12, 11, 10, 9, 7, 3, 2, 1] [12, 11, 10, 9, 6, 5, 4, 1] [12, 11, 10, 9, 6, 5, 2, 1]
[12, 11, 10, 8, 6, 5, 4, 2] [12, 11, 10, 8, 6, 4, 3, 1] [12, 11, 10, 7, 6, 5, 4, 1]
[12, 11, 10, 7, 5, 4, 3, 1] [12, 11, 9, 8, 7, 6, 4, 1] [12, 11, 9, 7, 6, 5, 4, 2]
[12, 11, 8, 7, 5, 4, 3, 2] [12, 10, 9, 8, 7, 5, 3, 2] [12, 10, 9, 8, 6, 5, 4, 2]

12 stadi, 10 taps: (2 combinazioni)
[12, 11, 10, 9, 8, 7, 5, 4, 3, 2] [12, 11, 10, 9, 8, 7, 5, 4, 3, 1]

Capitolo 2

Memorie e Matrici Logiche Programmabili

2.1 Memorie

2.1.1 Introduzione

Un utilizzo sempre più diffuso nei sistemi digitali stanno acquistando le memorie. Una memoria è un sistema, organizzato in celle elementari, dove è possibile immagazzinare, per tempi più o meno lunghi, informazioni digitali quali bits, bytes, etc.. Le memorie possono essere di tipo "sequenziale" o di tipo "ad accesso casuale". Un esempio di memoria sequenziale è il nastro magnetico. In una memoria di questo tipo occorre leggere tutta l'informazione memorizzata, in modo appunto sequenziale, prima di poter accedere a quella voluta. In una memoria ad accesso casuale è invece possibile accedere all'informazione desiderata specificando la locazione o posizione dove essa è stata scritta (ciò presuppone l'esistenza di un "registro" dove, per ciascun gruppo di dati, cioè per ciascun "file", sia stato preventivamente registrato l'indirizzo).

Un esempio di memoria ad accesso casuale è costituito dai dischi floppy o anche dai dischi rigidi (Hard Disks) dei computer. In questi la memorizzazione dell'informazione è basata su processi di magnetizzazione.

L'elaboratore di un computer ha bisogno anche di memorie, sia permanenti come le ROM (Read Only Memories o memorie a sola lettura) che temporanee come le RAM (Random Access Memories o memorie ad accesso casuale) che consentano tempi di accesso e trasferimento dati molto più veloci di quelli caratteristici dei dischi magnetici.

Le ROM incontrano le applicazioni più disparate, che vanno dall'immagazzinamento del codice (istruzioni) che vengono eseguite all'accensione di un calcolatore, alla memorizzazione di tabelle che forniscono il valore di funzioni predefinite (quali sin(x), cos(x), log(x) etc), alla memorizzazione di sequenze predefinite di impulsi e di forme d'onda, alla memorizzazione dei codici corrispondenti ai vari caratteri (codice BCD, codice EBCDIC, ASCII..) e via dicendo.

Le RAM sono memorie dinamiche, cioè il loro contenuto può essere modificato da programmi. Esse sono di grande utilità nella memorizzazione temporanea di sequenze o istruzioni generate dal codice che viene eseguito da un calcolatore.

2.1.2 Struttura di una ROM

La struttura della più diffusa tra le memorie statiche è quella della ROM. Illustriamo il suo funzionamento con un esempio concreto. Immaginiamo di voler effettuare rapidamente il calcolo del fattoriale (m) di un intero n (m=n!) con n minore di un valore prefissato. Possiamo effettuare il calcolo una volta per tutte, in corrispondenza ad ogni valore di n, e poi memorizzare il risultato in una ROM. Quando si abbia bisogno del fattoriale di un certo intero sarà sufficiente fornire in forma binaria tale numero ad una ROM come quella di figura 2.1 (dove, per motivi pratici, ci si è limitati ad interi ≤ 5). La ROM fornirà sui suoi 7 piedini d'uscita il fattoriale m richiesto.

Figura 2.1: Esempio di una struttura di ROM a 3 bit in ingresso e 7 in uscita.

È facile vedere che la ROM dovrà effettuare la conversione indicata nella tabella 2.1.

n	n_2	n_1	n_0	m	m_6	m_5	m_4	m_3	m_2	m_1	m_0
0	0	0	0	1	0	0	0	0	0	0	1
1	0	0	1	1	0	0	0	0	0	0	1
2	0	1	0	2	0	0	0	0	0	1	0
3	0	1	1	6	0	0	0	0	1	1	0
4	1	0	0	24	0	0	1	1	0	0	0
5	1	0	1	120	1	1	1	1	0	0	0

Tabella 2.1: Associazioni tra i numeri tra 0 e 5 ed i rispettivi fattoriali espressi in binario.

Tale conversione può essere, con riferimento all figura 2.2, effettuata nel modo seguente. Facendo uso di un decodificatore da 3 a 6 linee, si attiva, per un dato valore di n impostato in ingresso, una delle sei linee di uscita (indicate in figura con $w_0 \cdots w_5$). Ad esempio, in corrispondenza al codice 011 ingresso, si attiva la linea w_3.

Figura 2.2: Struttura che effettua la codifica di tabella 2.1.

A questa linea un apposito codificatore associa (cioè genera in uscita) il codice 0000110, cioè il valore decimale 6, che viene reso disponibile sulle linee ($m_0 \cdots m_6$). Analoga corrispondenza viene stabilita per gli altri valori applicati in ingresso, indicati nella tabella.

La struttura dei circuiti di decodifica e di successiva codifica può esser realizzata sulla falsa riga di quelle indicate rispettivamente nelle figure 2.3 e 1.32. Da notare che il circuito di figura 2.3 è relativo ad una decodifica da 4 a 10 linee, mentre nel

Figura 2.3: Circuito che effettua una decodifica da 4 a 10 linee.

Figura 2.4: Circuito che effettua la decodifica da 3 a 6 linee e la codifica.

nostro esempio è sufficiente un decodificatore da 3 a 6 linee. Il circuito di decodifica potrebbe essere quello mostrato nella parte sinistra della figura 2.4.

Infatti, è immediato verificare che le porte AND mostrate in figura implementano la funzione di *decoding* indicata nelle prime quattro righe della tabella. Ad esempio, per $n \equiv n_2, n_1, n_0 = 0, 1, 1$, si vede che è $W_3 = \overline{n}_2 \cdot n_1 \cdot n_0 \ = \ 1$, mentre $W_0 = W_1 = W_2 = W_4 = W_5 = 0$.

La parte destra della medesima figura svolge la funzione di codifica. Ad esempio possiamo vedere che, nel caso appena esaminato, in cui $W_3 = 1$, i diodi collegati alle linee (colonne) m_1 ed m_2, conducono, con che $m_1 = m_2 = 1$, mentre le rimanenti colonne rimangono collegate a massa (livello 0). Avremo quindi:

$$m \ = \ 0, 0, 0, 0, 1, 1, 0 \ = \ 6_{10}$$

che è appunto 3! È facile verificare il funzionamento in tutti gli altri casi.

Notiamo che l'architettura utilizzata per il decodificatore implica una porta AND per ciascuna linea (le linee vengono chiamate *word-lines*, le colonne *bit-lines*).

Tale architettura diviene costosa se il numero di word-lines è elevato. Per ovviare a tale problema, si può far uso di una decodifica del tipo di quella di figura 2.5, relativa al caso di una ROM da $512 \times 4 \ = \ 2048$ *bits* (codice in ingresso di 9 bit con codice d'uscita di 4 bit). Il decodificatore indicato nel blocco a sinistra in figura effettua la decodifica dei sei bit meno significativi, corrispondenti a 64 righe ($W_0 - W_{63}$). La matrice di codifica che segue ha 32 colonne (matrice di $64 \times 32 \ = \ 2048$ celle di memoria). Poiché in uscita sono richiesti quattro bit alla volta, i quattro blocchi selettori che seguono il codificatore, ciascuno con otto linee di ingresso, selezionano, per ciascun valore dei tre bit più significativi applicati in ingresso, una delle otto linee.

Figura 2.5: ROM da $512 \times 4 \ = \ 2048$ *bits* (codice in ingresso di 9 bit con codice d'uscita di 4 bit).

Tale tipo di indirizzamento è noto come indirizzamento X-Y o bidimensionale. Nell'esempio indicato sono necessarie solo 64 porte AND (o NAND) per la decodifica dei sei bit meno significativi. In più sono necessarie altre porte (36) per i selettori indicati. Il numero di porte richiesto è quindi pari a 100, da confrontare con le 512 necessarie nel caso di una struttura del tipo di quella indicata in figura 2.2.

Notiamo infine che, anche se per semplicità abbiamo discusso le matrici di codifica basandoci sull'esempio della matrice a diodi, questa è stata oramai sostituita da matrici in cui la funzione dei diodi è svolta da transistor o da MOSFET.

2.1.3 ROM programmabili

Le ROM che abbiamo descritto nella sezione precedente sono fornite, su specifica dell'utente, dai produttori. Le caratteristiche di tali ROM, cioè le informazioni in esse memorizzate, sono definite al momento della realizzazione e non possono essere alterate successivamente. È possibile far uso, in alternativa, di ROM programmabili dall'utente, note come PROM (field-programmable ROM). Una PROM, al momento della produzione, contiene tutte le possibili connessioni (diodi o transistor) interne. L'utente può, facendo uso di un'apposito hardware, bruciare o eliminare tutte le connessioni non volute, in modo da ottenere la tabella di conversione (corrispondenze ingressi-uscite) desiderata. L'hardware necessario è noto come "programmatore di PROM" . In una generica PROM, una volta effettuata la programmazione non è più possibile modificarla.

Esistono tuttavia delle PROM in cui la cancellazione dell'informazione e quindi la modifica della programmazione, è possibile. In questa varietà di PROM (note come EPROM o "erasable PROM") la cancellazione dei dati avviene per esposizione alla luce ultravioletta. Ciò richiede che il chip venga posto, per tempi relativamente lunghi (circa un'ora), in un apposito dispositivo dove sia presente una sorgente UV. Dopo la riprogrammazione, la parte sensibile della EPROM deve esser protetta (mediante un nastro adesivo che non trasmetta l'UV) per evitare che l'eventuale esposizione a radiazione UV (quale quella emessa dal Sole) possa cancellare l'informazione.

Per facilitare il processo di riprogrammazione sono state realizzate delle EPROM in cui la cancellazione dei dati viene effettuata con l'applicazione di una tensione dell'ordine di 10 V agli elementi che le costituiscono. Queste sono note come "electrically erasable PROM's" o anche $EEPROM$ o E^2PROM.

2.2 Memorie ad accesso casuale

Le ROM, come detto, sono utilizzate in tutte quelle applicazioni in cui sia necessario disporre di una memoria che conservi permanentemente l'informazione. In moltissime situazioni è invece necessario modificare di continuo l'informazione memorizzata. Sono quindi necessarie delle memorie che possano esser *scritte* oltre che *lette*. Questo è ad esempio il caso di un computer, che elabora dei dati e memorizza i risultati dell'elaborazione per una successiva stampa, trasmissione o nuova elaborazione. In tali casi si fa uso di *memorie ad accesso casuale*, note come RAM, o delle numerose varianti di queste.

La denominazione *accesso casuale* sta ad indicare il fatto che è possibile specificare un generico indirizzo in ingresso per avere in uscita la parola memorizzata a tale indirizzo. Ciò distingue questo tipo di memoria da un nastro magnetico, in cui occorre svolgere tutta la parte di nastro che precede quella dove la parola voluta è stata memorizzata. A rigore, anche la ROM è una memoria ad accesso casuale, nel senso appena detto per la RAM.

Esistono molti tipi di RAM diverse. Le SRAM (Static RAM) hanno come elemento di memoria il Flip-Flop. Esse conservano l'informazione memorizzata fino a che non si interrompa l'alimentazione. Le DRAM (Dinamic RAM) sfruttano come elemento di memoria la carica accumulata su di un dispositivo semiconduttore, inte-

grato in un chip. Tale cella di memoria è assimilabile ad un minuscolo condensatore, di dimensioni micrometriche. Le DRAM sono disponibili anche con un numero di celle di memoria molto elevato, ma hanno lo svantaggio di richiedere un continuo *refresh*, cioè debbono, ad intervalli di qualche millisecondo, essere *ricaricate*.

Esiste infine un tipo di RAM statica che conserva l'informazione quando l'alimentazione venga interrotta. Questa è nota come NVRAM o NVSRAM, sigla che sta per Non Volatile RAM. Essa nasce dall'unione di una RAM statica con una EEPROM, integrate sul medesimo chip.

Un esempio di RAM statica disponibile commercialmente è la P4C148 della Performance Semiconductor Corporation. Questa è una RAM da 4096 bit (organizzati in una matrice di 1024×4). La struttura del blocco funzionale è mostrata in figura 2.6, mentre la configurazione dei pin è mostrata in figura 2.7.

Figura 2.6: Struttura della RAM commerciale a 4096 bit P4C148.

Figura 2.7: Configurazione dei pin della RAM commerciale a 4096 bit P4C148.

L'indirizzo (10 bit) viene impostato sulle linee $A_0 - A_9$, mentre la parola (a 4 bit) da scrivere/leggere, viene impostata sulle linee $(I/O)_4 - (I/O)_1$.

Se l'ingresso indicato con $\overline{CE}/\overline{CS}$ (Chip Enable/Chip Select) è *alto*, l'uscita è in uno stato di "alta impedenza", qualunque siano i segnali sulle linee d'ingresso/uscita e sulla linea $\overline{WE}$ (Write Enable). In questo stato il consumo da parte del chip è fortemente ridotto. Se si vuole scrivere dati in memoria occorre mettere al livello *basso* l'ingresso $\overline{CE}/\overline{CS}$ come pure quello di $\overline{WE}$ dopo avere impostato i dati sugli ingressi I/O e l'indirizzo sugli ingressi A. Per leggere i dati, sempre mettendo *basso* l'ingresso $\overline{CE}/\overline{CS}$, occorre mettere alto l'ingresso $\overline{WE}$ (cioè disabilitarlo). Questa RAM è molto rapida (tempi di accesso compresi tra 10 e 25 ns).

Tra le molte applicazioni di una RAM, vale la pena di menzionare quella come *shift-register*. Occorrerà, come mostrato in figura 2.8 far uso di un contatore esterno per generare indirizzi successivi della RAM. Quando arriva un impulso di clock, il registro in ingresso (costituito da quattro FF di tipo D) carica i 4 bit in Q. L'invertitore fa si che il contatore agisca solo quando il segnale di clock va *basso*. Il sistema effettua quindi successivi shift di 4 bit in parallelo.

Figura 2.8: Utilizzo di una RAM come *shift-register*.

2.3 Matrici logiche programmabili

In parallelo alle PROM ed alle RAM, hanno avuto un rapido sviluppo negli ultimi decenni sistemi costituiti da matrici di porte logiche i cui elementi possono essere,

facendo uso di appositi programmi, opportunamente collegati tra loro in modo da implementare funzioni logiche complesse. Appartengono a tale categoria le PAL (Programmable Array Logic) le PLA (Programmable Logic Array) le FPGA (Field Programmable Gate Arrays) etc. In questa sezione ci occuperemo delle PLA.

Queste hanno molte similarità con le PROM, per cui può esser utile iniziare la discussione partendo dalla struttura della PROM. Questa, come si è visto, può essere schematizzata come un decodificatore accoppiato ad un codificatore. Il decodificatore è costituito da una matrice di porte AND, il codificatore da una matrice di OR (che negli esempi esaminati era realizzata con diodi). In un PROM ancora non programmata, tutte le connessioni delle porte AND e tutte quelle delle porte OR sono presenti. Si consideri ad esempio il circuito mostrato in figura 2.9, che rappresenta una matrice 2×2.

Figura 2.9: Esempio di PROM non ancora programmata: matrice 2×2 con tutte le connessioni presenti sia nel decodificatore, che nel codificatore.

Qui le crocette indicano i collegamenti esistenti. L'assenza di una crocetta starà a significare un collegamento interrotto. Il circuito di figura è una schematizzazione di quello reale. Ciascuna delle porte AND del circuito avrà in realtà quattro ingressi $(A, B, \overline{A}, \overline{B})$. L'eliminazione di uno o più dei collegamenti (operazione che nella EPROM può esser realizzata in laboratorio) implica l'eliminazione del relativo ingresso. Analoga osservazione va fatta per le porte OR, che sono, nel nostro esempio, a quattro ingressi, cioè ricevono le uscite delle porte AND 1,2,3,4 (almeno fino a che non siano eliminati uno o più dei relativi collegamenti).

Normalmente, in una PROM (o EPROM) il piano AND è fisso, cioè non tutti i collegamenti (le crocette nel grafico) sono presenti e quelli presenti non sono modificabili, mentre il piano OR contiene tutti i collegamenti. In una PLA sono presenti tutti i collegamenti del piano AND e tutti quelli del piano OR. In aggiunta, in una PLA sono presenti sul medesimo chip un certo numero di Flip-Flop di tipo D (registri) a cui sono collegate le uscite delle porte OR. Gli output di tali registri (nonché i loro complementi) sono disponibili, insieme ai normali pin di ingresso, come inputs per la matrice logica. Ciò consente di realizzare, oltre a logiche *combinatorie* anche logiche *sequenziali*.

Ammettiamo di voler realizzare, a partire dalla PLA di figura, un circuito (combinatorio) che fornisca:

$$Y_1 = A \oplus B$$

$$Y_0 = A + B$$

cioè l'OR esclusivo e l'OR. Per ottenere ciò è sufficiente, come à facile verificare, eliminare alcuni dei collegamenti, come mostrato nella figura 2.10.

Figura 2.10: PLA che realizza l'OR esclusivo per l'uscita Y_1 e l'OR per l'uscita Y_0.

Vediamo che con poche modifiche è possibile implementare funzioni logiche di

tipi diversi, a partire da un circuito dato. Un ulteriore esempio, relativo ad una matrice PLA 3×4 è mostrato in figura 2.11

Figura 2.11: Esempio di matrice PLA 3×4.

In aggiunta alle PLA sono disponibili in commercio le cosiddette PAL (Programmable Array Logic) ([1]). Queste differiscono dalle PLA per il fatto di avere la matrice delle porte AND programmabili mentre è fissata la matrice delle porte OR. Sia le PLA che le PAL sono disponibili sia in forma combinatoria (cioè senza registri) che in forma sequenziale (con registri).

2.3.1 Matrici per applicazioni sequenziali

La differenza tra la struttura combinatoria e quella sequenziale è di notevole importanza. Un circuito combinatorio fornisce infatti un'uscita che è funzione esclusivamente dei segnali presenti in quell'istante sui pin d'ingresso. Un circuito sequenziale invece fornisce in uscita, all'arrivo di un segnale di clock, dei livelli che sono funzioni *sia dei segnali presenti sui terminali d'ingresso che dei valori presenti alle uscite subito prima dell'arrivo dell'impulso di clock.*

Esaminiamo, come esempio di circuito sequenziale, il contatore in base 3 mostrato in figura 2.12. Indicheremo con l'indice 1 il FF di sinistra, con l'indice 2 quello di destra.

Figura 2.12: Esempio di circuito sequenziale: contatore in base 3.

Questo è un caso molto particolare di circuito sequenziale, poiché l'unico ingresso esterno è quello di clock. La porta NOR di figura riceve in ingresso l'uscita di entrambi i FF. Ammettiamo che lo stato iniziale del sistema sia quello con Q_1=0, Q_2=0. Con ciò avremo D_2=1 prima dell'arrivo del primo impulso di clock, come mostrato in tabella 2.2.

clock	D_1	D_2	Q_1	Q_2
	0	1	0	0
$\uparrow$	1	0	0	1
$\uparrow$	0	0	1	0
$\uparrow$	0	1	0	0

Tabella 2.2: Tabella delle verità per il contatore in base 3 mostrato in figura 2.12.

[1]Storicamente, le PLA hanno fatto la loro prima comparsa nel 1972, mentre le PAL sono divenute standard nel 1980. Le prime FPGA, che esamineremo dopo, sono state introdotte dalla Xilinx nel 1985

All'arrivo dell'impulso di clock, D_1 sarà trasferito in Q_1 (cioè Q_1=0) e D_2 in Q_2 (cioè Q_2=1). Avremo così la situazione mostrata nella seconda riga della tabella. Il successivo impulso di clock porta D_1 (che è 1) in Q_1, e D_2 (che è 0) in Q_2. La situazione sarà ora quella mostrata nella terza riga della tabella. Infine il terzo impulso di clock ripristina lo stato iniziale: Q_1=0, Q_2=0. Il conteggio ora riprende.

Lo schema di un sistema sequenziale, nel caso generale, è quello mostrato in figura 2.13.

Figura 2.13: Schema generale di un sistema sequenziale.

Le linee solide indicate nella figura costituiscono i "bus" che forniscono da un lato in input, alla parte combinatoria del circuito, le uscite dei registri (Flip-Flop) e dall'altro caricano nei registri le uscite delle porte logiche (gli OR).

In un circuito sequenziale il sistema passa attraverso una sequenza di stati o livelli dei suoi componenti. Se esso ha raggiunto un certo stato e se riceve da eventuali linee esterne opportuni impulsi, esso potrà passare ad uno di k possibili stati diversi, a seconda dello stato in cui esso è, e dei valori presenti sulle linee di ingresso esterne. Un sistema di questo tipo è descrivibile come una *macchina a stati finiti*. Esempi sono l'elettronica che controlla la distribuzione delle bibite in un distributore automatico, il sistema di controllo del ciclo in una lavabiancheria, di un semaforo etc.

Può accadere che un sistema di questo tipo venga a raggiungere uno *stato stabile*, cioè uno stato da cui esso non potrà più uscire. Tali situazioni, se si vuole che la macchina funzioni come programmato, vanno opportunamente evitate. Nell'esempio fatto prima, del divisore per 3, ciò non accade. In effetti, anche se il sistema si trovasse inizialmente nello stato $Q_1 = Q_2 = 1$, non previsto, dopo il successivo impulso di clock esso passerebbe nello stato $Q_1 = 0$, $Q_2 = 0$, dopo di che ricomincerebbe con la sequenza già vista.

Esaminiamo ancora, come esempio dell'uso di FF in circuiti sequenziali, un *temporizzatore*.

Ammettiamo che un certo strumento generi, in occasione di un evento esterno (l'accensione di un allarme, il passaggio di una particella, un brusco salto di tensione su di una linea) un impulso, e che questo impulso debba esser *letto* da un sistema "sincrono", cioè da un sistema digitale in cui tutte le transizioni tra stati hanno luogo in sincronia con un segnale di clock fissato. L'evento esterno avviene invece ad un istante casuale e non è quindi sincrono con tale clock. Se vogliamo che l'impulso generato possa agire sul sistema, dobbiamo in qualche modo "sincronizzarlo". Cioè dobbiamo,a partire da esso, generare un altro impulso che sia sincrono con l'impulso di clock che arriva subito dopo l'evento. Ciò può esser ottenuto facendo uso del circuito di figura 2.14, che fa uso di un FF.

Figura 2.14: Circuito sequenziale che funziona da temporizzatore.

L'evento esterno aziona, per un breve intervallo di tempo, un deviatore T, portandolo nella posizione A e riportandolo poi, dopo un tempo t_D (maggiore della

durata dell'impulso di clock) nella posizione B. Il sistema costituito dalle due porte NAND accoppiate genererà un livello alto sull'ingresso D del FF, di durata t_D. Tale segnale non arriverà però in Q, e quindi all'uscita della porta AND, se non all'arrivo del successivo impulso di clock (nel caso di figura, sul fronte di salita di questo). Vediamo così che l'impulso in uscita dalla porta AND è sincrono con il clock. Il funzionamento del circuito è visibile in figura 2.15.

Figura 2.15: Segnali di input ed output per il temporizzatore di figura 2.14.

È possibile realizzare un circuito sequenziale anche facendo uso di soli FF, cioè senza l'uso di porte logiche (AND, NAND etc.). Vediamo ad esempio un contatore sincrono in base 3, che fa uso di solo due FF JK, come mostrato in figura 2.16.

Figura 2.16: Circuito sequenziale, implementato con soli FF, che funziona come contatore sincrono in base 3.

È facile verificare, come mostrato nella tabella 2.3 ed in figura 2.17, il funzionamento del circuito.

ck	J_1	K_1	J_2	K_2	Q_1	$\overline{Q}_1$	Q_2	$\overline{Q}_2$
-	0	1	1	1	0	1	0	1
$\downarrow$	1	1	1	1	0	1	1	0
$\downarrow$	0	1	0	0	1	0	0	1
$\downarrow$	0	1	1	1	0	1	0	1

Tabella 2.3: Tabella delle verità relativa al contatore sincrono in base 3 di figura 2.16.

Vediamo infatti dalla tabella che dopo il terzo impulso (fronte di discesa) di clock, il sistema torna nello stato iniziale. Esaminiamo più in dettaglio il comportamento. Se lo stato iniziale è quello con $Q_1 = Q_2 = 0$, avremo $\overline{Q}_1 = 1$. Vediamo allora che è $J_2 = K_2 = 1$, mentre $J_1 = Q_2 = 0$. IL primo impulso di clock trova il secondo FF pronto ad eseguire un cambiamento di stato (toggle), mentre l'uscita del primo sarà ancora 0. Avremo così: $J_1 = Q_2 = 1$, $\overline{Q}_2 = 0$, $Q_1 = 0$, $\overline{Q}_1 = J_2 = K_2 = 1$. Il secondo impulso di clock troverà che ora il primo FF è in condizione di "toggle", come continua ad esserlo il secondo. La transizione causata dal clock porterà quindi a: $Q_1 = 1$, $Q_2 = 0$ e quindi: $J_1 = Q_2 = 0$, $Q_1 = 1$, $\overline{Q}_1 = 0$. Ora il secondo FF non è più in condizione di "toggle", nè lo è il primo. La transizione causata dal clock porterà allora a $Q_1 = 0$, $\overline{Q}_1 = 1$, mentre Q_2 e $\overline{Q}_2$ rimangono inalterati (essendo $J_2 = K_2 = 0$). Abbiamo così l'ultima riga della tabella, identica alla prima. Il ciclo ricomincia quindi dopo tre cicli di clock.

Figura 2.17: Segnali di input ed output relativi al contatore sincrono in base 3 di figura 2.16.

2.4 Gli integrati FPGA

Gli FPGA (Field Programmable Gate Arrays) sono stati introdotti nel 1985 dalla Xilinx Inc.. Essi hanno incontrato una larghissima diffusione grazie alla loro grande versatilità e relativa facilità d'uso. Essi, al pari delle PAL, PLA, EPROM, EEPROM, possono essere programmati in modo da rispondere a tutto un campo di applicazioni particolari. Esistono in commercio FPGA con strutture e caratteristiche diverse tra loro. Come il nome ci dice, si tratta di circuiti programmabili "sul campo", vale a dire che non occorrono *programmatori* appositi, come nel caso delle EPROM.

Questi integrati contengono milioni di transistor, che realizzano un elevato numero di circuiti logici i quali possono essere collegati tra loro facendo uso di commutatori programmabili. Gli FPGA sono una varietà di PLD (Programmable Logic Devices). Tuttavia, a differenza dei PLD, gli FPGA non contengono porte AND ed OR. Al loro posto essi contengono dei blocchi logici costituiti da "Lookup Tables", che vengono adoperati per implementare funzioni logiche. Tali blocchi logici sono accessibili tramite blocchi di input/output (I/O) che corrono lungo la periferia del chip e sono connessi tra loro per mezzo di blocchi di interconnessione, come è possibile vedere in figura 2.18. Una tipica FPGA contiene da circa 10,000 a milioni di elementi equivalenti a porte logiche, con interconnessioni programmabili dall'utente, oltre a centinaia di migliaia di celle di memoria. Una qualsiasi funzione logica, sia di tipo combinatorio che sequenziale, puo' essere implementata facendo uso delle intrinseche strutture logiche quali le Celle Logiche (LC) o Blocchi Logici (LB). Le LC (LB) sono come delle piccole ROM, in cui sono immagazzinate delle Lookup Tables (LUT) che contengono i valori delle funzioni logiche elementari, per ogni combinazione di valori impostati in ingresso.

Figura 2.18: Struttura a blocchi di un FPGA.

La figura 2.19 mostra un po più in dettaglio la struttura di una tipica FPGA. Si possono vedere le celle logiche, le interconnessioni, le celle di I/O e la memoria. Si vede anche il dettaglio di una tipica cella logica, costituita in questo esempio da tre "generatori di funzioni logiche" (basati su lookup-tables) ed un FF di tipo D.

La varietà delle "architetture" disponibili in commercio è enorme. Esse differiscono sia nelle dimensioni che nella struttura delle LC e delle interconnessioni. Alcune LC si riducono all'equivalente di porte NAND a due ingressi, altre hanno una struttura più complessa e possono esser dotate di Multiplexers e di Look-Up Tables (LUT). In alcune FPGA un blocco logico corrisponde ad una intera struttura del tipo di una PAL. In ogni caso, una caratteristica comune alla più gran parte delle architetture è la presenza di FF di tipo D in ciascuna LC.

Figura 2.19: Struttura a blocchi di un FPGA, con il dettaglio di una cella logica.

2.4.1 Gli FPGA della Xilinx

Descriveremo ora, a titolo di esempio, le caratteristiche di una FPGA prodotta dalla Xilinx.

Tale FPGA ha tre elementi configurabili principali: i Blocchi Logici Configurabili (CLB), i Blocchi di Input-Output (IOB) e le Risorse Programmabili di Interconnessione IPR. I CLB costituiscono gli elementi funzionali per costruire la logica voluta dall'utente. Gli IOB forniscono l'interfaccia tra i pins dell'integrato e le linee interne di segnale. Le IPR forniscono i percorsi di collegamento per connettere gli ingressi e le uscite dei CLB e degli IOB alle reti appropriate. È possibile programmare le celle di memoria statica interna per determinare le funzioni logiche da svolgere come pure i collegamenti esterni dell'FPGA.

La figura 2.20 mostra una parte dei blocchi logici e delle relative interconnessioni di un FPGA. Tutte le connessioni interne sono composte di segmenti metallici con punti di contatto programmabili, per implementare i collegamenti desiderati. Le IPR sono sovrabbondanti, in modo da consentire un'implementazione efficiente dei collegamenti. Vi sono quattro tipi diversi di interconnessioni, tre delle quali si distinguono solo per la lunghezza dei segmenti: linee di lunghezza singola, di lunghezza doppia e linee lunghe. In aggiunta esistono otto linee globali, adoperate per clocks o segnali di controllo globali.

Figura 2.20: Blocchi logici ed interconnessioni di un FPGA.

Un esempio di CLB è quello mostrato in fig 2.21. Ciascun CLB contiene una coppia di FF e due generatori di funzioni logiche indipendenti (indicati con F e G nella figura), ciascuno dei quali ha quattro ingressi. Questi generatori di funzioni logiche hanno un elevato grado di flessibilità, grazie appunto alla presenza di quattro ingressi (la più gran parte delle porte logiche combinatorie hanno meno di quattro ingressi). Tuttavia, un terzo generatore di funzioni logiche (H) è aggiunto, come mostrato in figura. Il generatore H ha due ingressi. Uno o entrambi questi ingressi possono essere le uscite di F o G; l'altro (o entrambi) possono essere esterni al CLB. Ne segue che il CLB può implementare funzioni di un numero di variabili fino ad un massimo di 9.

I CLB implementano la parte principale della logica di questo FPGA. La flessibilità e la simmetria dell'architettura dei CLB facilitano il posizionamento ed i collegamenti in una qualsiasi applicazione.

La figura 2.22 mostra più in dettaglio la struttura interna del CLB. I due FF possono essere adoperati per immagazzinare i segnali in uscita dai generatori di funzioni logiche. Tuttavia i FF ed i generatori di funzioni logiche possono anche essere adoperati indipendentemente. L'ingresso indicato con DIN può essere adoperato come un input diretto per ciacuno dei due FF. Il segnale H1 può pilotare l'altro FF attraverso il generatore H. Le uscite dei generatori di funzioni logiche sono anche

disponibili all'esterno del CLB, facendo uso di due outputs indipendenti da quelli dei FF. Tale versatilità aumenta la densità di componenti logici e semplifica i collegamenti.

Figura 2.21: Schema interno di un Blocco Logico Configurabile (BLC) di un FPGA.

Figura 2.22: Dettaglio della struttura interna di un Blocco Logico Configurabile (BLC) di un FPGA.

2.4.2 L'FPGA della ALTERA

La ALTERA produce FPGA con un numero di celle logiche (chiamate Logic Elements dalla ALTERA) che va da circa 500 ad oltre 12000. Concentreremo la nostra attenzione sul modello denominato FLEX 10K. La sigla FLEX sta per Flexible Logic Element Matrix, ed indica una struttura in cui i LE sono a loro volta organizzati in blocchi di matrici logiche (Logic Array Blocks) o LAB (vedasi la figura 2.23). Ciascun LAB contiene otto LE. Gli elementi di un dato LAB sono collegati tra loro da una connessione locale (quindi veloce) detta Local Interconnect. Ciascun LE consiste di una LUT a quattro ingressi, un FF programmabile, oltre ad un certo numero di linee dedicate. Gli otto LE di un LAB possono essere adoperati per creare blocchi logici di medie dimensioni (contatori a 8 bit, decodificatori d'indirizzo o macchine di stato). Più LAB possono esser uniti insieme per creare blocchi di complessità ancora maggiore. Ciascun LAB rappresenta l'equivalente di 96 porte logiche effettive.

Come si vede in figura 2.23, sono presenti sul chip anche un certo numero di Embedded Array Blocks (EAB) che sono costituiti essenzialmente da FF, Multiplexers e RAM. Essi possono essere adoperati per implementare ROM, RAM, Registri FIFO, funzioni logiche etc.. Nel caso della serie FLEX 10K, ciascun EAB può contribuire con l'equivalente di 100-600 porte logiche, per realizzare funzioni logiche complesse (moltiplicatori, microcontrollori, macchine di stato etc.). Ciascun EAB può esser adoperato indipendentemente, o diversi EAB possono esser combinati per implementare funzioni di maggior complessità.

Figura 2.23: Struttura di un FPGA dell'ALTERA.

La figura 2.24 mostra in dettaglio la struttura di un EAB.

Figura 2.24: Struttura di un Embedded Array Block (EAB) di un FPGA dell'ALTERA.

Come accennato, questo è un blocco flessibile di RAM, dotato di registri sia alle porte d'ingresso che a quelle d'uscita. Esso può essere adoperato per implementare

quelle funzioni logiche di maggiore complessità note come "megafunzioni". Ad esempio un EAB può essere adoperato per programmare moltiplicatori, filtri digitali, circuiti di correzione di errori etc.. Le funzioni logiche effettuate dal EAB vengono implementate, al momento della programmazione, costruendo delle LUT. Ciò rende molto più veloci i tempi di calcolo. Un EAB può anche essere adoperato come RAM, con una struttura di 256 parole di 8 bit, 512 da 4, 1024 da 2 o 2048 da 1. Inoltre più EAB possono esser combinati insieme per realizzare dei blocchi di RAM di dimensioni maggiori.

Un software dedicato aiuta l'utente a realizzare tutte le operazioni volute. Nei pacchetti software forniti sono già implementate moltissime delle funzioni logiche, comprese quelle di maggiore complessità.

La figura 2.25 mostra la struttura di un insieme di LE, cioè di un LAB.

Ciascun LAB consiste di otto LE, nonché dei relativi ingressi e delle interconnessioni locali. Inoltre esso fornisce quattro segnali di controllo (con inversioni programmabili) che possono essere adoperati da tutti e otto gli LE del blocco. Due di questi ingressi possono essere adoperati come clocks; gli altri due come ingressi di controllo di Clear/Preset.

Figura 2.25: Struttura di un LAB (formato da otto LE) di un FPGA dell'ALTERA.

La figura 2.26 mostra il dettaglio di un LE. Questo costituisce la più piccola delle unità logiche nell'architettura di questo FPGA. Ciascun LE contiene una LUT a quattro ingressi, che costituisce un generatore di funzioni logiche attraverso il quale è possibile calcolare una generica funzione logica di quattro variabili. In aggiunta, esso contiene un FF programmabile, dotato di un Enable sincrono, e connessioni da/per altri elementi analoghi. Il FF può esser programmato come un JK, SR, D etc.. Gli ingressi di clock, Clear e Preset sul FF possono esser pilotati da segnali esterni globali, da segnali di I/O, o da altra logica interna.

Figura 2.26: Dettaglio della struttrura di un LE di un FPGA dell'ALTERA.

Capitolo 3

Sistemi Logici Complessi

3.1 Introduzione

In questo capitolo tratteremo sistemi logici costituiti da insiemi di porte logiche, Flip-Flop ed, in alcuni casi, memorie. Tra questi rientrano quelle che sono note come "macchine a stati finiti", che trovano numerosissime applicazioni pratiche. Esse sono adoperate in sistemi automatici di apertura/chiusura di accessi, in macchine per distribuzione di bevande o altro, nei videogiochi, nei telefonini, semafori etc.

Passeremo poi ad esaminare la struttura base dei microprocessori e microcontrollori. Tra questi ultimi analizzeremo in dettaglio, a titolo di esempio, un particolare modello: il 16F877 della Microchip®.

Data l'importanza che essi hanno sia nella struttura interna dei processori, che nella comunicazione con periferiche, discuteremo anche la struttura dei diversi tipi di BUS adoperati.

3.2 Macchine a stati finiti

Un semplice esempio concreto che si presta ad illustrare la struttura a blocchi di una macchina a stati finiti è offerto dalla barra automatica che consente l'accesso ai binari di una metropolitana. La figura 3.1 mostra lo schema a blocchi di una tale macchina, limitatamente alla struttura base.

Figura 3.1: Schema a blocchi della macchina a stati finiti che controlla l'apertura e chiusura della sbarra d'accesso ad una metropolitana.

La macchina ha solo due stati: "bloccato" e "sbloccato". Quando essa è nel primo di questi stati una persona può, inserendo un biglietto, sbloccarla. La macchina passa ora nello stato "sbloccato", come mostrato nel disegno dalla linea che indica la transizione causata dall'*azione* "immissione del *biglietto*".

L'etichetta associata alla transizione ha due parti, separate da uno "/". La prima di queste indica l'*evento* che causa la transizione, la seconda il nome dell'*azione* che sarà eseguita una volta che l'evento si sia verificato. L'azione porta il sistema nello stato "sbloccato". Analogamente vediamo che all'evento "passaggio" segue l'azione "blocco", che porta il sistema nello stato bloccato.

Dobbiamo, per completezza, vedere come descrivere situazioni anomale che possono verificarsi. Ad esempio, cosa accade se qualcuno cerca di passare senza aver inserito il biglietto? Possiamo prevedere che in tal caso la barra rimanga bloccata, e scatti un allarme. L'altra situazione anomala che potremmo prevedere è che la barra sia già aperta e la persona inserisca un secondo biglietto. Le modifiche che potrebbero esser apportate al precedente schema a blocchi, per ovviare a queste situazioni anomale sono illustrate in figura 3.2.

Figura 3.2: Schema a blocchi modificato della macchina a stati finiti che controlla l'apertura e chiusura della sbarra d'accesso ad una metropolitana.

Vediamo dalla figura che nel primo dei due casi ipotizzati lo stato del sistema non cambia (la sbarra rimane bloccata) ma scatta un segnale d'allarme. Nel secondo, la sbarra rimane sbloccata, ma il passeggero si sente rivolgere un "grazie"!

Se il passeggero è passato senza aver introdotto un biglietto, la sbarra potrebbe esser stata danneggiata. In tal caso un semplice allarme può non esser sufficiente. Potremmo voler far passare il sistema in un nuovo stato, che definiremmo "violazione" e farlo rimanere in tale stato fino all'intervento di un tecnico. La figura 3.3 illustra le modifiche apportate con l'introduzione di tale nuovo stato.

Figura 3.3: Aggiunta di uno stato "violazione" allo schema a blocchi della macchina a stati finiti che controlla l'apertura e chiusura della sbarra d'accesso ad una metropolitana.

Notiamo che ora il sistema può uscire dallo stato "violazione" a seguito di un *evento* "pronto", con che esso torna nello stato normale "bloccato". Abbiamo inserito anche un nuovo evento particolare ("disattiva") che il tecnico può generare per spegnere l'allarme mentre procede al controllo ed all'eventuale riparazione. Abbiamo aggiunto altresì un nuovo stato (uno pseudo-stato, indicato da un pallino nero) in cui la macchina si trova quando è spenta. L'accensione fa passare la macchina nello stato iniziale (bloccato).

L'esempio presentato, estremamente semplice (ma che può divenire estremamente complicato nella realtà) illustra i concetti fondamentali che sono alla base di una macchina a stati finiti (cui per brevità ci riferiremo nel seguito come FSM, per Finite State Machine). I concetti base sono quelli di "stato" di "azione" e di "transizione" tra stati.

Esaminiamo in dettaglio un esempio tratto dall'elettronica digitale: un circuito che verifica la parità di una serie di bit. L'output del circuito dovrà essere *alto* se in ingresso è arrivato un numero dispari di "1", *basso* nel caso contrario.

Definiamo "stato dispari" del circuito quello associato alla prima situazione, "stato pari" il secondo. La figura 3.4 mostra gli stati e le transizioni tra di essi.

Figura 3.4: Schema a blocchi di una FSM per la generazione del bit di parità di una stringa di bits.

Vediamo dal diagramma che se il sistema è nello stato "pari" ed il prossimo bit in arrivo è un "1", esso passa nello stato "dispari". Da questo, se il bit successivo è ancora un "1", il sistema ritorna nello stato "pari". Se invece il bit è uno "0", il sistema non cambia stato, come indicato dalle linee chiuse (bit=0). All'interno del simbolo di ciascuno dei due stati abbiamo evidenziato in parentesi quadra l'output del sistema.

Il diagramma di figura 3.4 si traduce facilmente in una tabella delle verità. Questa è la 3.1.

Stato Attuale	Bit in ingresso	Stato Successivo	output
pari	0	pari	0
pari	1	dispari	0
dispari	0	dispari	1
dispari	1	pari	1

Tabella 3.1: Tabella delle verità per il diagramma a blocchi del generatore del bit di parità di figura 3.4.

Notiamo che l'output è quello associato allo *Stato Attuale*, non allo *Stato Successivo*. Indicando con l'abbreviazione StA lo Stato Attuale e con StS lo Stato Successivo, e definendo "0" lo stato pari ed "1" quello dispari, la precedente tabella diviene la 3.2.

Stato Attuale	Bit in ingresso	Stato Successivo	output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Tabella 3.2: Tabella delle verità modificata per il diagramma a blocchi del generatore del bit di parità di figura 3.4.

Da questa tabella vediamo che:

$$StS = StA \oplus Bit$$

Tale funzione è di conseguenza facilmente implementabile facendo uso di un OR esclusivo e di un FF di tipo D, come mostrato in figura 3.5.

Figura 3.5: Circuito che implementa una FSM per la generazione del bit di parità di una stringa di bits.

Il circuito è di tipo sincrono, cioè il cambiamento di stato ha luogo, dopo che il Bit sia stato fornito all'ingresso dell'XOR, in corrispondenza al fronte di salita dell'impulso di clock. L'uscita dell'XOR definisce lo Stato Successivo, mentre il Q del FF fornisce la parità associata allo Stato Attuale.

Un'analisi della tabella consente di proporre un'implementazione alternativa della medesima macchina. Notiamo infatti che se l'input (Bit) è uguale a "0", lo stato non cambia con l'arrivo dell'impulso di clock. Se l'input è invece "1", lo Stato Successivo è il complemento di quello Attuale. Ciò suggerisce di far ricorso ad un FF di tipo JK, come mostrato in figura 3.6.

Figura 3.6: Circuito che implementa una FSM per la generazione del bit di parità di una stringa di bits, con l'uso di un FF di tipo JK.

Anche in questo caso è importante che il Bit in ingresso sia stabile prima del fronte di salita dell'impulso di clock.

3.2.1 Temporizzazioni nelle FSM

Si preferisce in genere che una macchina di stato funzioni in modo *sincrono*, cioè che tutti i FF che la compongono effettuino il passaggio da uno stato a quello successivo all'arrivo del medesimo impulso di clock.

Come accennato nell'ultimo esempio esaminato, occorre che tutti gli inputs ed outputs siano *stabili* all'arrivo dell'impulso di clock. La *validità* degli output dovrà essere assicurata dopo il fronte di salita (o di discesa) del clock.

Per chiarire tali concetti, esaminiamo l'esempio di due FSM comunicanti, come quelle mostrate schematicamente in figura 3.7.

Figura 3.7: Due FSM che comunicano tra loro. La macchina FSM1 riceve in ingresso l'output Y della FSM2 e viceversa.

Ammettiamo che entrambe le macchine siano sincrone, con transizione sul fronte di salita del clock e che l'output X della FSM1 sia l'input della FSM2, mentre l'output Y della FSM2 sia l'input della FSM1.

Ammettiamo inoltre che entrambe le macchine abbiano due soli stati, A e B per la FSM1, C e D per la FSM2. Ciascuna delle due macchine inoltre cambi stato se al suo ingresso è presente, all'arrivo del segnale di clock, un livello *alto*, mentre rimanga nello stato precedente se il segnale all'ingresso è *basso*. La figura 3.8 mostra il segnale di clock, la successione degli stati in cui ciascuna delle due macchine verrà a trovarsi ed i segnali X ed Y alle uscite.

Figura 3.8: Successione degli stati e forme d'onda alle uscite di due FSM comunicanti. La forma d'onda in alto è il clock comune alle due macchine.

La FSM1 sta per entrare nello stato A all'arrivo del secondo impulso di clock, mentre la FSM2 è nello stato C, con l'uscita Y bassa. Durante il secondo ciclo di

clock, la FSM1 è nello stato A, con X alto, mentre la FSM2 è in C, con Y basso. Con il terzo impulso di clock l'input Y alla FSM1 è ancora basso, quindi essa rimane nello stato A. L'input X alla FSM2 è alto, il che fa si che essa passi nello stato D. Con il quarto impulso di clock la FSM1, che ora ha l'ingresso Y alto, passa nello stato B, mentre l'ingresso X della FSM2 è divenuto alto da troppo poco tempo perché essa possa effettuare la transizione; quindi essa rimane nello stato D. Ora X è basso mentre Y è alto. Con il quinto impulso di clock, la FSM1 passa nello stato A mentre la FSM2 rimane in D. Con il sesto impulso di clock, essendo sia X che Y alti, entrambe le macchine cambiano stato, passando rispettivamente in B ed in C.

3.2.2 FSM per il controllo di una macchina distributrice

Consideriamo un problema lievemente più complicato di quelli esaminati finora: quello di una FSM per il controllo di una macchina che, dopo l'inserimento di una certa somma, eroga una qualche bevanda. Ammettiamo che il costo della bevanda sia 15 centesimi e che il pagamento possa esser effettuato in monetine da 5 e da 10 centesimi. Indicando con "C" la moneta da 5 centesimi e con "D" quella da 10, la sequenza delle monetine da immettere potrà essere:

1. C, C, C
2. C, C, D
3. D, C
4. D, D
5. C, D

Il secondo ed il quarto caso corrispondono a situazioni in cui la persona ha erroneamente immesso più del necessario.

Il corrispondente diagramma degli stati è mostrato in figura 3.9.

Figura 3.9: Diagramma degli stati della FSM per il controllo di una macchina per la distribuzione di bevande.

Non abbiamo indicato le transizioni che non implicano un cambiamento di stato. Ad esempio, nello stato S_0, se non è stata immessa alcuna moneta, la macchina dovrà restare nel medesimo stato, anche se ciò non è indicato in modo esplicito.

Il diagramma può esser semplificato, tenendo conto del fatto che gli stati S_4, S_5,S_6,S_7,S_8 corrispondono al medesimo stato (erogazione). Possiamo semplificare ulteriormente lo schema di figura, pensando a ciascuno stato come definito dalla somma totale immessa fino a quell'istante. Cioè, lo stato S_1 corrisponde a "5c", gli stati S_2 ed S_3 a "10c", etc. Si può allora ridisegnare il diagramma come mostrato in figura 3.10.

Vediamo che ora gli stati sono solo quattro. Inoltre la transizione dallo stato "10c" allo stato "15c" può esser causata sia dall'azione "C", che dalla "D" (cioè dall'OR delle due). Dall'ultimo diagramma si ottiene poi la tabella delle verità 3.3.

Figura 3.10: Diagramma degli stati semplificato della FSM per il controllo di una macchina per la distribuzione di bevande.

Stato	*Input*		*Stato*	*Output*
Attuale	D	C	*Successivo*	
0c	0	0	0c	0
0c	0	1	5c	0
0c	1	0	10c	0
0c	1	1	15c	0
5c	0	0	5c	0
5c	0	1	10c	0
5c	1	0	15c	0
5c	1	1	15c	0
10c	0	0	10c	0
10c	0	1	15c	0
10c	1	0	15c	0
10c	1	1	15c	0
15c	X	X	15c	1

Tabella 3.3: Tabella delle verità per il diagramma a blocchi di figura 3.10.

Notiamo che è stato previsto il caso in cui siano state immesse contemporaneamente sia la moneta da 5c che quella da 10c. Inoltre, l'erogazione è associata allo stato "15" (anche se raggiunto con l'immissione di una somma più alta !).

Associamo ora agli stati "0c", "5c", "10c", "15c" le configurazioni di due bit, rispettivamente uguali a:

$$0c \rightarrow 00$$

$$5c \rightarrow 01$$

$$10c \rightarrow 10$$

$$15c \rightarrow 11$$

La tabella acquista allora la forma 3.4.

Dobbiamo ora realizzare il circuito che implementa tale macchina. Per farlo, cominciamo con il minimizzare la tabella facendo uso delle mappe K per ciascuna delle funzioni D_1, D_0, *Output*. Le mappe ottenute sono mostrate nelle figure 3.11, 3.12, 3.13.

Figura 3.11: Mappa di Karnaugh per la variabile D_1, ottenuta dalla tabella 3.4.

Stato Attuale		*Input*		*Stato Successivo*		*Output*
Q_1	Q_0	D	C	D_1	D_0	*(erogazione)*
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
0	1	1	1	1	1	0
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	0	1	1	1	1	0
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	X	X	1

Tabella 3.4: Forma modificata della tabella delle verità per il diagramma a blocchi di figura 3.10.

Figura 3.12: Mappa di Karnaugh per la variabile D_0, ottenuta dalla tabella 3.4.

Figura 3.13: Mappa di Karnaugh per la variabile *Output*, ottenuta dalla tabella 3.4.

Dall'analisi delle mappe si ottiene:

$$D_1 = Q_1 + Q_0 \cdot C + D$$

$$D_0 = C \cdot \overline{Q}_0 + Q_0 \cdot \overline{C} + Q_1 \cdot C + Q_1 \cdot D + D \cdot C$$

$$output = Q_1 \cdot Q_0$$

È poi immediato verificare che un circuito che implementa queste funzioni è quello di figura 3.14.

Figura 3.14: Circuito che realizza le funzioni riassunte nella tabella 3.4.

3.3 Microprocessori, Microcontrollori e Microcomputer

3.3.1 Introduzione

Ci siamo occupati finora di porte logiche, memorie, flip-flop e, più in generale, di sistemi costituiti da combinazioni di porte logiche, flip-flop, registri/memorie.

È chiaro che, con una scelta opportuna dei componenti, è possibile costruire sistemi che realizzino una qualsivoglia funzione logica. Abbiamo esaminato alcuni di tali sistemi: quelli che effettuano operazioni di somma, sottrazione, moltiplicazione e divisione di numeri a più bit, quelli che calcolano la parità di un numero binario, etc. Risulterebbe tuttavia assai pesante se si dovesse costruire un nuovo circuito per ogni nuovo problema. Non è di oggi l'esigenza di sistemi versatili che, opportunamente "programmati" si prestino a realizzare molte operazioni, anche di tipo molto diverso. In questi sistemi, il problema di realizzare strutture che implementino funzioni di varia natura, si traduce in quello di fornire al sistema una sequenza di istruzioni da eseguire; quello che appunto chiamiamo un "programma".

I sistemi che eseguono tali sequenze di istruzioni posseggono un ulteriore grado di flessibilità: essi possono eseguire una certa sequenza prestabilita di istruzioni, ma eseguirne, a partire da un certo punto, una diversa se il risultato ottenuto a quel punto ha una determinata caratteristica.

Si supponga ad esempio di voler far eseguire la sequenza di istruzioni che segue:

1. si legga un numero N
2. si verifichi se N è positivo
3. se N è positivo, se ne calcoli la radice quadrata
4. se N è negativo, se ne modifichi il segno e si calcoli poi la radice quadrata
5. si stampi il risultato, attribuendogli il segno del numero N originale e si fermi l'esecuzione

In questa sequenza vediamo che all'istruzione 2 può seguire la 3 se N è positivo, mentre seguiranno le 4, 5 se N è negativo.

L'esempio fatto, estremamente banale, illustra come l'esecuzione di sequenze diverse possa essere condizionata dal verificarsi di determinati *eventi*. L'*evento* era nel nostro esempio costituito dall'esser negativo il segno del numero dato. In un caso generale l'evento può esser costituito dal risultato ottenuto, in una serie di operazioni aritmetiche consecutive, ad un certo punto della sequenza. Ad esempio in una sequenza di operazioni che calcola il rapporto incrementale di una certa funzione $f(x)$:

$$R(x_0, \Delta x_i) = \frac{f(x_0 + \Delta x_i) - f(x_0)}{\Delta x_i}$$

per valori decrescenti di Δx_i, l'evento può semplicemente esser costituito dal verificarsi della condizione:

$$\frac{|R(x_0, \Delta x_i) - R(x_0, \Delta x_{i-1})|}{|R(x_0, \Delta x_i)|} < \epsilon$$

con ϵ una costante data.

È anche possibile, in un caso ancor più generale, che l'evento abbia un'origine esterna. Si supponga ad esempio di voler realizzare un dispositivo che mantenga la temperatura di un certo ambiente attorno ad un valore prestabilito T_0, alimentando un condizionatore che può anche funzionare da pompa di calore. Il nostro dispositivo sarà collegato, oltre che al condizionatore, ad uno strumento di misura della temperatura.

Le istruzioni dovranno prevedere che:

1. Il sistema procede ad una prima lettura della temperatura

2. Se la temperatura è maggiore di T_0, il condizionatore inizierà a raffreddare l'ambiente

3. Il sistema procederà a leggere, ad intervalli di tempo regolari, la temperatura

4. Se la temperatura diviene minore di T_0, il condizionatore, funzionando ora da pompa di calore, inizierà a riscaldare l'ambiente

5. Il sistema procederà a leggere, ad intervalli di tempo regolari, la temperatura

6. Se la temperatura diviene maggiore di T_0 viene nuovamente eseguito il ciclo di istruzioni a partire dalla 2 (viene effettuato quello che chiameremo un *JUMP*).

I dispositivi che eseguono tali funzioni sono i microcontrollori. Di questi esiste una grande varietà. Essi possono differire per velocità, numero di connessioni con il mondo esterno, tipo di programmazione, etc.. Il cuore di un microcontrollore è il microprocessore.

Un tipico microprocessore non dispone di periferiche (tastiera, connessione a stampanti, etc.; in alcuni casi neanche memorie). Un *microcontrollore*, come anche un *microcomputer* è sostanzialmente un microprocessore opportunamente integrato da periferiche, collegate al microprocessore da apposite linee di comunicazione (BUS). Un microcontrollore è in genere integrato in un singolo chip ed è dotato di un numero relativamente elevato di porte di I/O. Queste possono essere adoperate sia per leggere parametri fisici esterni forniti da opportuni sensori, sia per controllare l'operazione di dispositivi esterni.

Ci occuperemo in questa sezione della struttura base di un microcontrollore. Passeremo poi, nella successiva, ad esaminare in dettaglio l'architettura di un microcontrollore particolare: il *16F877* della *microchip*. Discuteremo poi le istruzioni base riconosciute da tale microcontrollore, e le illustreremo infine con un esempio concreto. Non discuteremo in modo esplicito l'operazione di un microcomputer, anche se la più gran parte delle considerazioni che faremo, riferite ai microcontrollori, sono applicabili anche ai microcomputer.

3.3.2 Struttura base di un tipico microcomputer e di un microcontrollore

Un diagramma semplificato di un tipico microcomputer è mostrato in figura 3.15.

Figura 3.15: Schema di massima di un microcomputer.

Al cuore del microcomputer è un microprocessore. Questo riceve vari tipi di dati dall'esterno, tra cui troviamo:

Le istruzioni che servono al microprocessore per poter iniziare il proprio lavoro: tipicamente gli "indirizzi" su cui poter leggere il programma da eseguire; gli "indirizzi" dove scrivere ("indirizzi" di display e stampanti) le informazioni in uscita, l'indirizzo della tastiera e di eventuali sensori o pulsanti, etc. Vedremo tra poco cosa intendiamo per "indirizzi". Tutte le comunicazioni tra il microprocessore ed il mondo esterno avvengono attraverso appositi collegamenti elettrici (o in alcuni casi ottici) costituiti in genere da gruppi di otto o più linee elettriche dedicate, dette BUS. Nell'esempio di figura un BUS (unidirezionale) collega una ROM con il microprocessore, un secondo BUS (bidirezionale) collega il microprocessore con una memoria dinamica (RAM), ulteriori BUS collegano il microprocessore con i dispositivi di input/output (I/O).

Esaminiamo ora la struttura interna del microprocessore che costituisce la parte "intelligente" del nostro sistema. Questa è mostrata sommariamente in figura 3.16. Al cuore del sistema è la CPU (Central Processing Unit).

Figura 3.16: Schema di massima di un microprocessore.

Le memorie possono essere interne o esterne al microprocessore. La maggior parte dei microprocessori oggi in uso hanno una certa quantità di memorie, integrate insieme alla CPU in un unico chip.

La CPU contiene una propria logica di controllo ed un certo numero di registri (costituiti tipicamente da gruppi di 8/16/32 celle di memoria veloce in cui vengono caricate sia le istruzioni che debbono essere eseguite che i dati numerici). Programmi e dati risiedono nelle memorie. La CPU preleva le istruzioni dalle memorie e le esegue, prelevando dalle stesse memorie anche gli eventuali dati necessari.

Una forma lievemente modificata della struttura mostrata nella precedente figura è quella di figura 3.17.

Figura 3.17: Schema del microprocessore. Sono indicati i registri, la logica di controllo ed il bus degli indirizzi.

L'informazione binaria è trasferita, attraverso i BUS indicati, dalla memoria nei registri e verso la logica di controllo, ed adoperata come segue:

1. L'informazione trasferita alla logica di controllo costituisce l'*istruzione*. Essa viene tradotta dalla logica di controllo in opportune sequenze di *micro-operazioni*, richieste dall'istruzione. Ad esempio, l'istruzione potrebbe richiedere di trasferire un dato (byte o parola di più bytes) dalla memoria in uno dei registri.

2. I dati necessari ad eseguire l'istruzione vengono trasferiti dalla memoria nei registri, sotto il controllo della logica

In realtà il trasferimento di istruzioni e di dati dalla memoria ai registri ed alla logica di controllo, utilizza il medesimo BUS. In effetti uno schema che meglio corrisponde al caso che concretamente si incontra è quello di figura 3.18.

Figura 3.18: Schema del microprocessore. Sono indicati i registri, la logica di controllo ed il bus degli indirizzi. Notiamo che ora un unico BUS connette tutti i componenti.

Vediamo da tale schema che i dati provenienti dalle memorie viaggiano lungo le linee del BUS il quale è collegato sia ai registri che all'unità contenente la logica di controllo. Analogamente, le istruzioni generate dalla logica di controllo sono "viste" sia dalle memorie che dai registri. Ciò creerebbe delle ambiguità se non fosse stata introdotta una particolare struttura per il BUS. Questo, nel caso più semplice, viene suddiviso in un insieme di più BUS separati:

(a) il DATA BUS, lungo il quale viaggiano i dati. Questo è un BUS bidirezionale, avente un numero di linee uguale al numero di bit di cui è costituita la parola del processore. Ad esempio 8 o 16 o 32 linee.

(b) l'ADDRESS BUS (BUS degli indirizzi) che definisce l'indirizzo (cella di memoria o periferica) a cui la parola impostata sul DATA BUS è destinata

(c) il CONTROL BUS (o STROBE) che può, in processori diversi, assolvere funzioni diverse. Ad esempio in alcuni computer questo BUS ha due linee: un livello alto su una delle due linee segnala la richiesta che il dato proveniente sul DATA BUS venga *scritto* all'indirizzo di periferica specificato dall'ADDRESS BUS; un segnale basso segnala invece una richiesta di *lettura* del dato dall'indirizzo di periferica specificato. La seconda linea di STROBE è invece adoperata per segnalare la richiesta di lettura o scrittura del dato dalla *memoria*, all'indirizzo specificato sulle ADDRESS LINES.

Alcune osservazioni sono a questo punto necessarie.

1. In primo luogo vediamo che il BUS, indicato in modo molto sommario nella figura, collega in realtà un numero molto elevato di indirizzi. Esso deve infatti consentire l'accesso ad ogni singola cella di memoria, il che fa sì che ad esempio, in un processore avente una memoria di 64 kbyte occorra un numero di linee nell'ADDRESS BUS almeno pari a 16; in uno di 4 Gbyte almeno pari a 32.

2. Quando un componente immette dati (cioè livelli di tensione) sulle DATA LINES, occorre che tutti gli altri componenti risultino "sganciati" dalle medesime linee. Ciò è ottenuto in genere utilizzando, per il collegamento alle DATA LINES, delle porte logiche "tri-state": si tratta di porte le cui uscite, oltre che negli ovvi stati "HIGH" e "LOW", possono essere in un terzo stato in cui l'impedenza verso massa è molto grande, stato noto appunto con il nome di "HIGH IMPEDANCE"[1]. In un qualsiasi istante, l'uscita di uno solo degli elementi collegati al DATA BUS può essere in uno dei due stati logici; tutte le altre saranno nel terzo stato. Questa osservazione non vale per le ADDRESS LINES, poiché su queste solo l'unità di controllo può scrivere

3. Nella più gran parte dei processori, le istruzioni eseguite dai diversi componenti sono "sincrone". Un apposito CLOCK, di frequenza fissata, determina l'intervallo di tempo in cui un ciclo di operazioni ha inizio e fine. Ad esempio, in alcuni processori la lettura di un byte dalla memoria richiede un ciclo di CLOCK, l'esecuzione della somma di 2 byte può richiedere 6 cicli di CLOCK, etc..

4. In una macchina in cui il DATA BUS ha una larghezza pari a 8 la lettura di una parola di 32 bit (4 bytes) richiederà quattro letture consecutive dalla memoria.

5. Quelli che genericamente abbiamo chiamato "dati" (nel caso di una macchina con un DATA BUS di larghezza otto, un singolo byte) possono indifferentemente essere numeri o istruzioni. Infatti un'istruzione altro non è che una sequenza di bit. Nel nostro esempio della macchina avente parole di 8 bit, possiamo avere quindi $2^8 = 256$ istruzioni elementari. Vedremo in seguito alcuni esempi di tali istruzioni.

6. L'architettura sommariamente descritta, in cui un unico BUS viene adoperato sia per i dati che per i programmi, è nota come "architettura di Von Neumann". Essa è adoperata in processori di uso generale ed implica che la memoria utilizzata per i dati e quella adoperata per i programmi debbano avere parole di uguale lunghezza. Un'architettura alternativa, molto frequente nei microcontrollori, è quella "Harvard". In tale architettura sono presenti due BUS separati, uno per i dati ed uno per i programmi. In tal caso, le rispettive memorie possono esser costituite da parole di lunghezza diversa. Ad esempio, la memoria utilizzata per i dati (RAM) può avere parole di 8 bit, mentre quella adoperata per i programmi può aver parole di 12, 14, 16 o 32 bit. Il vantaggio di tale architettura consiste nel fatto che la CPU può effettuare contemporaneamente un accesso alla memoria dati ed uno alla memoria programmi, riducendo in tal modo il tempo di esecuzione. Vedremo in seguito un esempio di microcontrollore basato su tale architettura.

Possiamo ora vedere in dettaglio le operazioni che la CPU esegue per effettuare operazioni di I/O. Nel caso in cui sia la CPU a inviare dati verso una cella di memoria

[1] Un esempio di integrato tri-state è il latch trasparente 74125, già incontrato nella sezione 1.15.2

o una periferica, l'indirizzo di questa viene impostato sulle ADDRESS LINES, mentre i dati vengono caricati sulle DATA LINES. Successivamente la CPU provvede a caricare un segnale di STROBE su una delle CONTROL LINES, per segnalare che i dati sono pronti. Quando il "destinatario" (cella di memoria o periferica) vede il proprio indirizzo sulle ADDRESS LINES esso provvede a copiare l'informazione contenuta sulle DATA LINES in un proprio buffer. Se viceversa la CPU deve leggere dei dati dalle celle di memoria o da una periferica, essa pone l'indirizzo di questa sulle ADDRESS LINES e carica un segnale di STROBE sulle opportune CONTROL LINES. La periferica "vede" il proprio indirizzo sulle ADDRESS LINES, verifica la presenza del segnale di STROBE relativo ad un'operazione di lettura sulle CONTROL LINES e carica allora i dati sulle DATA LINES. Fatto ciò, la medesima periferica resetta la linea di STROBE, a segnalare che i dati sono pronti.

Affinché la periferica possa confrontare l'indirizzo presente sulle ADDRESS LINES con il proprio, questa dovrà far uso di un opportuno comparatore ad N bit, dove N è la larghezza del BUS degli indirizzi. A tale scopo si fa uso di opportuni *decodificatori d'indirizzo*, alcuni esempi dei quali sono discussi nella prossima sezione.

Il protocollo di comunicazione descritto, apparentemente semplice è in realtà incompleto, come vedremo più avanti.

3.3.3 Decodificatori d'indirizzo

Un decodificatore d'indirizzo è un dispositivo che riceve in ingresso il BUS degli indirizzi, confronta i singoli bit con quelli, memorizzati in precedenza dall'utente, che definiscono l'indirizzo del particolare dispositivo o cella di memoria, e fornisce in uscita un opportuno livello logico che attivi il dispositivo o cella di memoria.

Un integrato che è a volte utilizzato per realizzare tale operazione è il 74LS85, mostrato in figura 3.19. Questo è un semplice comparatore a 4 bit, che fornisce un livello logico alto sul piedino indicato con $A = B$, se si ha uguaglianza dei 4 bit A con i 4 bit B. Sugli ingressi si potranno ad esempio inviare i segnali del BUS (supposto di larghezza 4) mentre gli ingressi B saranno collegati ai livelli definiti dall'utente nello specificare l'indirizzo del dispositivo. Se il bus ha larghezza superiore a 4, si potrà far uso di più integrati 74LS85, collegati in parallelo.

Figura 3.19: Un comparatore a 4 bit, utilizzabile per realizzare un decodificatore d'indirizzo a 4 linee.

In modo analogo si può far sì, come mostrato in figura 3.20, che la CPU selezioni in scrittura una data porta di I/O, utilizzando un decodificatore d'indirizzo ottenuto mettendo assieme più 74LS85 (4 nell'esempio di figura) in associazione ad un latch ottale 74LS273 e ad alcune porte NAND. L'arrivo del segnale di STROBE trasferirà il contenuto del BUS dati sull'uscita del latch, solo se l'indirizzo presente sulle ADDRESS LINES è quello giusto.

Figura 3.20: Decodifica d'indirizzo e trasmissione dei dati ad un'unità di I/O.

Un altro tipo di decodificatore d'indirizzo spesso adoperato è il 74ALS679, il cui layout è mostrato in figura 3.21. Questo è essenzialmente una porta NAND a 12 ingressi, dei quali un numero programmabile può essere invertito. Il numero degli ingressi da invertire, contati a partire da A1, è determinato dal codice binario presente sugli ingressi P=P3,P2,P1,P0. Ad esempio, se poniamo P=1010, avremo che i primi 10 ingressi saranno invertiti. Ammettiamo ad esempio di avere un BUS di indirizzi a 12 linee, e di voler attribuire ad una cella di memoria o porta di I/O l'indirizzo binario 111111110000 (esadecimale EE0) porremo: P3=0,P2=1,P1=0,P0=0. Con ciò, se sul BUS indirizzi è presente la configurazione suddetta, l'uscita della porta sarà *bassa*. Ciò segnalerà che il dispositivo associato è quello selezionato (se il livello basso è quello adoperato per la selezione; altrimenti occorrerà far seguire alla porta un NOT).

Figura 3.21: Chip NAND a 12 ingressi, per il quale un numero programmabile di ingressi può essere invertito. Il numero di ingressi invertiti (contati a partire da A1) è determinato dai bit di comando P3,P2,P1,P0.

3.3.4 I/O programmato: registri di stato, interrupts e DMA

Le procedure di Input/Output esaminate finora prevedevano che l'intero processo fosse controllato dalla CPU. Questa è una restrizione eccessiva; infatti in tal modalità, il dispositivo di I/O è fermo ad aspettare che la CPU si occupi di leggere i dati da esso immessi sulle DATA LINES. Consideriamo ad esempio una semplice tastiera; se noi pigiamo uno dei tasti, dovremo aspettare che la CPU abbia letto il corrispondente codice dalle DATA LINES, prima di batterne un'altro. Non abbiamo però alcun modo di sapere se la CPU ha effettuato la lettura. Se inoltre il dispositivo di I/O adoperato (nastro magnetico, disco..) è intrinsecamente molto veloce, le sue prestazioni risulteranno drasticamente peggiorate, a causa della necessità che esso aspetti di volta in volta il segnale di OK da parte della CPU.

Si può ovviare a tale problema facendo uso dei cosiddetti "registri di stato". Questi consentono al dispositivo di I/O di "segnalare" alla CPU che vi sono dei dati pronti ad esser letti. Ad esempio, nel caso della tastiera appena menzionato, quando noi battiamo un tasto, un qualche circuito collegato alla tastiera dovrà settare un bit su di una certa linea, per segnalare alla CPU che è presente sulle DATA LINES un carattere pronto ad esser letto. La CPU dovrà verificare di continuo lo stato della linea contenente il bit di stato (in questo caso il "registro" ha un solo bit) e tutte le volte che il bit è alto procederà alla lettura. La stessa CPU provvederà poi a resettare il bit, indicando in tal modo alla periferica che il dato è stato letto. A volte (ad esempio nel caso della tastiera) il bit di stato può esser trasmesso attraverso una delle DATA LINES; basterà sceglierne una che non sia utilizzata per i dati veri e propri.

L'operazione di set e poi di reset del bit di stato, può ad esempio esser effettuata tramite un FF, come mostrato schematicamente in figura 3.22.

Quando viene battuto un tasto, l'informazione della tastiera viene caricata sulle linee d'ingresso del registro 74ACT574. La tastiera genera anche un impulso posi-

Figura 3.22: Protocollo di comunicazione tra una tastiera ASCII e la CPU.

tivo che, arrivando all'ingresso di clock del FF, fà passare questo nello stato di *set* (Q=1). Il medesimo impulso, arrivando sull'ingresso di clock del 74ACT574, attiva il trasferimento del dato (carattere) sul BUS DATI. Il buffer tri-state '125, con l'ingresso collegato al Q del FF e l'uscita collegata alla linea 7 del BUS, pone tale linea a livello alto. La CPU vede tale livello ed inizia il processo di lettura del carattere sul BUS DATI. Essa invia un poi un impulso negativo all'ingresso OE del 74ACT574, nonché al CLEAR del FF, con che sia l'uscita Q del FF che la linea D7 del DATA BUS ritornano al livello 0, a segnalare che un nuovo dato può esser immesso.

Nell'esempio fatto, il registro di stato è di un solo bit. Nel caso più generale tale registro avrà un certo numero di bit, ciascuno dei quali segnalerà una diversa condizione. Ad esempio, oltre al bit che indica la presenza di un dato da leggere, ci potrà esserne uno che indica, nel caso della lettura di un nastro magnetico, l'inizio o la fine del nastro; un'altro che segnali un'errore di parità, etc..Tutti questi bit saranno impacchettati in una STATUS WORD (un byte) che la CPU dovrà leggere per testare poi i singoli bit.

Anche se l'utilizzo dei registri di stato è di enorme utilità nelle operazioni di I/O, risulta necessario nella maggioranza dei casi far uso anche dei cosiddetti *interrupts*. Questi sono dei segnali (spesso *active low*, o *edge triggered*) generati da una periferica per richiamare l'attenzione della CPU. Infatti, con il semplice uso degli STATUS REGISTERS, accade che la CPU deve spendere un'elevata frazione del proprio tempo ad interrogare (operazione detta *polling*) i vari dispositivi e verificare le rispettive STATUS FLAGS. Se il numero dei dispositivi è elevato, la CPU finirebbe col non fare altro. Ciò è illustrato in figura 3.23. L'utilizzo degli *interrupts* risolve tale problema.

Figura 3.23: Protocollo di interrogazione dei registri di stato (polling).

A ciascun dispositivo, o gruppo di dispositivi, può esser attribuito un *interrupt* che esso attiva per richiedere l'attenzione della CPU. L'interrupt generato fa sì che la CPU interrompa la propria azione per verificare l'origine ed il motivo dell'interruzione. Ciò che la CPU fa in tal caso è completare l'istruzione in corso, salvare nello STACK quella successiva e nei registri eventuali dati; poi identificare il dispositivo che ha generato l'interrupt e leggere il relativo registro di stato, per decidere ciò che deve esser fatto.

In pratica, quando la CPU vede un interrupt su di una certa linea (le linee di interrupt sono spesso indicate con le sigle IRQ: IRQ1, IRQ2, IRQ3...) essa, dopo aver completato l'istruzione in corso e memorizzato le altre informazioni sopra descritte, salta ad un indirizzo di memoria prestabilito (uno per ogni IRQ) dove è memorizzata la procedura (*interrupt handler*) che deve esser eseguita per quel particolare interrupt. Tale procedura altro non è che una routine, scritta spesso in Assembler, che effettua i compiti prestabiliti legati al particolare interrupt. Gli indirizzi degli

interrupt handlers sono memorizzati in quello che è noto come l'*interrupt vector*. Uno schema che descrive sommariamente la gestione degli interrupts è mostrato in figura 3.24.

Figura 3.24: Protocollo di gestione degli interrupts.

Una complicazione in tale procedura deriva dal fatto che spesso, ad un dato interrupt sono collegati più dispositivi di I/O. Per risolvere il problema, i collegamenti vengono realizzati nel modo schematicamente indicato in figura 3.25.

Figura 3.25: Protocollo di gestione degli interrupts, nel caso in cui più dispositivi siano connessi al medesimo IRQ.

Come si vede, alla linea IRQ2 sono collegati, tramite un buffer tri-state, sia il dispositivo A che il C. Se solo uno di questi ha generato un interrupt (portando la linea corrispondente al livello *basso*), l'uscita del buffer dell'altro dispositivo sarà nello stato di alta impedenza; ne segue che il livello di tensione presente sulla linea sarà basso.

Ammettiamo ora che la CPU "veda" un'interrupt su di una certa linea di interrupt, ad esempio la IRQ2. Essa, come già detto, completerà l'esecuzione dell'istruzione in corso, salverà i dati nei registri e passerà il controllo allo "handler" associato al determinato interrupt (IRQ2 nel nostro esempio). Ora la routine "handler" sa quale è l'interrupt che lo ha attivato, ma non sa quale dei due dispositivi associati a tale linea lo abbia generato. Per scoprirlo, lo handler legge i *registri di stato* di tutti i dispositivi connessi al determinato livello di interrupt (infatti, il dispositivo che genera l'interrupt deve sempre specificare la richiesta settando uno o più *status bits*). Una volta che lo handler abbia individuato il dispositivo che ha effettuato la richiesta ed anche, dallo status bit, il *tipo* di richiesta, esso esegue l'azione necessaria e, in aggiunta, resetta lo *status bit* (o gli *status bits*) e l'IRQ generato. Notiamo per inciso che, se più di un dispositivo aveva attivato il determinato IRQ, anche se viene disattivato il buffer tri-state del primo dispositivo (cioè tale buffer viene messo nello stato di alta impedenza) rimane attivo (cioè basso) quello del secondo.

Lo handler restituisce ora il controllo alla CPU. Questa, qualora veda il determinato IRQ ancora attivo, chiama nuovamente lo handler che a sua volta và a leggere i *registri di stato* dei dispositivi connessi al determinato livello di interrupt. Ora troverà che il registro associato al primo dispositivo è resettato mentre è ancora attivo quello relativo al secondo. Lo handler esegue ora l'azione richiesta da tale dispositivo, resetta lo (o gli) *status bit*, resetta il determinato IRQ e restituisce il controllo alla CPU.

A volte è opportuno impedire che un dispositivo, normalmente abilitato a generare interrupts, ne generi. Ad esempio, una stampante potrebbe generare interrupts tutte le volte che il suo buffer è vuoto, per richiedere altri dati da stampare. Se tuttavia la stampa è terminata, non ha senso che la stampante continui a generare

interrupts, che potrebbero creare problemi, in particolare nel caso in cui la stampante condivida con altri dispositivi il medesimo livello di interrupt. Si procede allora ad una disabilitazione della capacità della stampante di generare interrupts (non si disabilita tuttavia il particolare IRQ, che potrebbe esser adoperato da altri dispositivi).

Il "polling" cioè la ripetuta interrogazione dei registri di stato, può esser paragonato al sollevare il telefono ogni minuto per verificare se qualcuno ci stia chiamando. L'uso degli interrupts può esser invece paragonato con l'aspettare che suoni il telefono, prima di sollevare il ricevitore.

L'utilizzo dei registri di stato, o anche degli interrupts, risulta troppo lento nei casi in cui si debba trasferire rapidamente grandi quantità di dati da un dispositivo verso la CPU o verso un'altro dispositivo. Ad esempio, nel caso in cui si debba leggere o scrivere dati su di disco o nastro magnetico, o acquisire dati da un sensore esterno, a velocità di diverse centinaia di migliaia di bytes/secondo, sarebbe impensabile applicare il sistema descritto sopra, con l'uso di interrupts per ciascun byte trasferito. Quello che si fà in questi casi è di far si che l'unità periferica esterna prenda direttamente il controllo dei BUS; tale unità si comporta, a partire da un certo istante, da BUS Master, comportandosi come una CPU. Essa, una volta abilitata dalla CPU, pone sulle ADDRESS LINES l'indirizzo del destinatario, sulle DATA LINES i dati da trasferire e sulle CONTROL LINES le istruzioni di lettura/scrittura.

3.3.5 Operazione della logica di controllo ed esecuzione delle istruzioni

La logica di controllo altro non è che un piccolo circuito sequenziale. La sua funzione è quella di generare i segnali di controllo per le singole operazioni elementari, quali quelle di trasferire i dati dalla memoria ad appositi registri, di modificare singoli dati (e.g. effettuare uno *shift* dei bit in un byte, calcolare l'AND di un byte con un altro, etc.) di fornire i dati all'unità logico aritmetica (ALU) che è quella sezione della CPU in grado appunto di effettuare operazioni aritmetiche o logiche.

Gli inputs all'unità di controllo sono i bit dell'istruzione che deve essere eseguita; ciascuno dei bits di ogni istruzione particolare farà sì che l'unità di controllo attraversi in modo ciclico un'unica sequenza di stati, allo scopo di generare le singole micro-operazioni di cui un'istruzione elementare è costituita. Una tipica sequenza è mostrata nella figura 3.26.

Figura 3.26: Sequenza delle istruzioni in un microprocessore e micro-operazioni generate da ciascuna istruzione.

Durante la prima parte del ciclo, l'istruzione è letta dalla memoria e fornita agli inputs della logica di controllo. Tale operazione è effettuata dalla logica di controllo in modo automatico per una qualsivoglia istruzione. Questo è noto come "ciclo di prelevamento" (FETCH).

La seconda parte è il ciclo di esecuzione vero e proprio. Questo dipende ovviamente dalla particolare istruzione. Una volta che la logica di controllo abbia generato

i segnali di controllo richiesti dall'istruzione, essa automaticamente esegue un altro ciclo di FETCH, per leggere l'istruzione successiva dalla memoria, dopo di che il ciclo continua.

3.3.6 Struttura interna di una semplice macchina

Una rappresentazione schematica, un pò più dettagliata di quelle viste finora, di un semplice microprocessore è quella mostrata in figura 3.27.

Figura 3.27: Struttura interna di un tipico microprocessore.

Tutti i registri sono collegati attraverso una BUS comune. Anche se in genere non tutti i registri avranno la medesima dimensione (numero di bit) il BUS che li collega dovrà avere una larghezza pari alla dimensione del più largo di essi. Le funzioni svolte dai diversi tipi di registri sono elencate qui di seguito:

1. MEMORY ADDRESS REGISTER (MAR). Non è altro che il registro responsabile di scrivere sulle ADDRESS LINES l'indirizzo di memoria da accedere in lettura/scrittura.

2. MEMORY DATA REGISTER (MDR). Questo è il registro responsabile di porre sulle DATA LINES i dati da scrivere in memoria o leggere dalla memoria.

3. INSTRUCTION REGISTER (IR). Questo è il registro in cui viene memorizzata temporaneamente l'istruzione che si esegue. Questo registro è l'interfaccia diretta con la logica di controllo; il "codice di operazione" (OPCODE) viene caricato nell'IR durante il ciclo di FETCH ed è poi adoperato dalla logica di controllo per generare le successive micro-operazioni necessarie ad implementare l'istruzione.

4. Il PROGRAM COUNTER (PC). Questo è il registro contenente l'indirizzo in memoria della prossima istruzione da eseguire. È anche noto come INSTRUCTION POINTER (IP).

5. L'ACCUMULATORE (A). È il registro in cui la più gran parte dell'elaborazione del dato viene effettuata. Esso costituisce uno degli input della ALU.

6. Il CONTROL REGISTER (CR). È quello che fornisce i segnali di STROBE alle CONTROL LINES. Esso consente alla logica di controllo di sincronizzare il flusso dei dati tra la CPU e le memorie.

7. Il DATA POINTER (DP). È un registro che contiene l'indirizzo dell'operando (dato) richiesto da una determinata istruzione

8. Lo STACK (non mostrato nella figura 3.27). Questo è un insieme di registri, ciascuno di dimensione pari a quella del BUS degli indirizzi di programma. È un dispositivo di tipo "Last In-First Out" (LIFO), come mostrato in figura 3.28.

Figura 3.28: Struttura ed operazioni dello STACK.

Nello STACK possono essere memorizzati consecutivamente indirizzi di istruzioni. Un comando di PUSH carica la nuova istruzione in cima allo STACK, spingendo le altre verso il basso. Un commando di POP, estrae l'ultima istruzione dalla sommità dello STACK e sposta tutte le altre verso l'alto. Lo STACK è comunemente adoperato nelle *chiamate* a sottoprogrammi. Quando una tale chiamata (CALL) ha luogo la CPU pone nella prima locazione libera dello STACK l'istruzione che segue il CALL (cioè quella che sarebbe stata eseguita in assenza del CALL), salta poi alla locazione dove è memorizzata la prima istruzione del sottoprogramma, esegue questa e le successive ed infine, quando incontra l'istruzione che termina il sottoprogramma (normalmente un RETURN), esegue un POP, cioè carica dalla sommità dello STACK l'indirizzo che era stato memorizzato al momento del CALL. Tale procedura può esser ripetuta più volte in sequenza nel caso in cui il sottoprogramma chiami a sua volta altri sottoprogrammi. Si vede da ciò che un parametro importante è la *profondità* dello STACK (cioè il numero di registri che lo costituiscono.

9. Lo STACK POINTER (SP). È un registro dedicato adoperato per indicare la posizione nello STACK della successiva istruzione da eseguire.

10. PROCESSOR STATUS WORD (PSW). È una parola contenente le *flags* che indicano i codici di stato, generati dal processore.

11. I GENERAL REGISTERS (B,C,D,E). Questi sono registri d'uso generale adoperati dalla CPU. Hanno la medesima larghezza dell'accumulatore.

3.3.7 Le istruzioni in un tipico microprocessore

Ciascuna istruzione inviata alla CPU comporterà in qualche modo l'elaborazione o manipolazione di dati all'interno della CPU: trasferimento di dati, somma, confronto, etc.. A tale scopo ciascuna istruzione dovrà contenere i seguenti cinque elementi:

1. **L'operazione da eseguire** (somma, trasferimento di dati, caricamento, immagazzinamento,etc.). Questa parte dell'istruzione costituisce il *codice operativo* (OPCODE). Un tipico microprocessore disporrà di un set molto limitato di operazioni primitive.

2. **La sorgente dei dati**, cioè l'indirizzo dove trovare i dati. Se i dati sono in memoria occorrerà specificarne l'indirizzo; se i dati sono altrove la loro locazione dovrà essere specificata in altro modo.

3. **La sorgente del secondo dato**, per quelle istruzioni che ne richiedano uno (la somma richiede due dati mentre il complemento ne richiede solo uno).

4. **La destinazione del dato risultante** dopo che l'operazione sarà stata eseguita.

5. **La locazione dell'istruzione successiva** da eseguire dopo che quella attuale sia stata completata.

Chiaramente tutte queste informazioni non possono essere contenute in una singola locazione di memoria (tipicamente 8 bits) e di conseguenza è necessario ridurre in qualche modo la lunghezza dell'istruzione. La riduzione è ottenuta facendo opportune ipotesi su alcune di queste:

- l'informazione 5 può essere omessa facendo l'ipotesi che l'istruzione successiva da eseguire segua immediatamente l'istruzione corrente in memoria (cioè che le istruzioni siano immagazzinate in locazioni consecutive della memoria)
- Le informazioni 2, 3, 4 possono essere eliminate facendo uso dei registri interni alla CPU per immagazzinare i dati (o l'indirizzo dei dati) e rendendo l'uso di tali registri implicito nella particolare istruzione. Ciò aumenta il numero di istruzioni possibili, ma attraverso una scelta giudiziosa, è possibile mantenere il numero di istruzioni al di sotto di 256, in modo tale che esse possano essere rappresentate da un singolo byte.

Ad esempio nel processore 8085, nell'istruzione:

ADD C (istruzione=81H, o 10000001 in binario)

è implicito che:

1. L'operazione è una somma
2. La sorgente del primo byte di dati è l'accumulatore (implicito nell'istruzione)
3. La sorgente del secondo byte di dati è il registro C (implicito nell'istruzione)
4. La destinazione è l'accumulatore (implicito nell'istruzione)

L'istruzione completa richiede così solo 8 bits e può quindi esser contenuta in una singola locazione di memoria.

Dopo aver ridotto in tal modo la dimensione delle istruzioni, è tuttavia ancora necessario per alcune istruzioni fornire ulteriori informazioni in aggiunta al codice operativo. Una tale istruzione comporterà tipicamente un accesso alla memoria e sarà quindi necessario fornire l'indirizzo della locazione di memoria da leggere. In tal caso è necessario far uso di più di otto bits per specificare l'istruzione; una tale istruzione vien detta **istruzione multibyte**.

Il primo byte di un'istruzione **istruzione multibyte** è l'OPCODE. I bytes addizionali, che forniscono l'informazione aggiuntiva formano ciò che vien detto l'*operando*. L'intera sequenza dei bytes costituisce l'istruzione.

3.3.8 I microcicli

L'esecuzione di ciascuna istruzione nella CPU comporta l'esecuzione di un certo numero di operazioni ridotte dette micro-operazioni o microcicli. Ciascun microciclo

impiega un ciclo di clock e rappresenta un'operazione di trasferimento dati o di manipolazione di registri. La notazione adoperata è tipicamente la seguente:

Operazione di trasferimento dati:

$$destinazione \leftarrow sorgente$$

$$MAR \leftarrow PC$$

$$IR \leftarrow MDR$$

La prima di queste istruzioni copia il Program Counter nel Memory Address Register. La seconda copia il Memory Data Register nell'Instruction Register.

Operazione sui registri:

$$destinazione \leftarrow espressione$$

$$PC \leftarrow PC + 1$$

$$SP \leftarrow SP - 1$$

La prima di queste istruzioni incrementa di uno il Program Counter; la seconda decrementa di uno lo Stack Pointer.

Qualora sia necessario trasferire dati alla memoria esterna o leggere dati da tale memoria, ciò non può esser fatto in un singolo microciclo. La memoria esterna è considerata come una serie di registri analoghi ai registri interni della CPU; tuttavia prima che uno di questi registri (parola) possa essere letto o scritto, è necessario indicare quale sarà il registro coinvolto. Ciò vien fatto mettendo l'indirizzo della parola nel MEMORY ADDRESS REGISTER. Ciò fa si che l'appropriata parola di memoria sia abilitata e collegata al MEMORY DATA REGISTER. I dati possono quindi essere trasferiti tra il MDR e la parola indirizzata (la direzione del trasferimento dipende dalla circostanza che si tratti di un'azione di lettura o di scrittura). Tale trasferimento di dati deve aver luogo almeno un ciclo di clock dopo che l'indirizzo della parola è stato scritto sul MAR, in modo da dar tempo alla memoria di stabilizzarsi. Questi micro cicli sono rappresentati come segue:

$$MAR \leftarrow registro\ sorgente$$

$$MDR \leftarrow memoria(MAR)$$

Ciascun microciclo che comporti un trasferimento di dati tra due dei registri interni della CPU fà uso del BUS interno; di conseguenza solo uno di questi microcicli può aver luogo durante un singolo ciclo di clock. Se tuttavia un microciclo comporta una manipolazione di un singolo registro o usa la memoria esterna o i BUS esterni allora esso può aver luogo simultaneamente ad un trasferimento tra registri.

3.3.9 Assembler e linguaggi di livello più elevato

Come visto nella precedente sezione, le istruzioni che un processore è in grado di eseguire sono estremamente elementari, il che fa sì che per realizzare operazioni che ci

appaiono banali, quale ad esempio elevare un numero ad una certa potenza occorrerà fornire al processore un numero relativamente grande di istruzioni elementari.

Per chiarire ulteriormente tale punto, esaminiamo ancora un esempio, relativo al vecchio microprocessore 6502. Le istruzioni elementari necessarie per sommare due numeri (consistenti ciascuno di un singolo byte esadecimale) in linguaggio macchina sono elencate nella tabella 3.5.

Il programma in linguaggio macchina sarebbe allora:

&70 = &A5
&71 = &80
&72 = &18
&73 = &65
&74 = &81
&75 = &85
&76 = &82
&77 = &60

istruzione	*tipo*	*significato*
A5	codice operativo	pone nell'accumulatore il contenuto della locazione che segue
80	operando	indirizzo della locazione di memoria in cui è contenuto il primo addendo
18	codice operativo	azzera il riporto
65	codice operativo	somma al contenuto dell'accumulatore il dato contenuto all'indirizzo indicato dalla locazione di memoria che segue
81	operando	indirizzo della locazione di memoria dove è memorizzato il secondo operando
85	codice operativo	memorizza il contenuto dell'accumulatore all'indirizzo specificato dall'operando seguente
82	operando	locazione di memoria in cui deve essere immagazzinata la risposta
60	codice operativo	fine di questa sezione del programma

Tabella 3.5: Istruzioni elementari necessarie per sommare due numeri in linguaggio macchina.

Qui il simbolo & che precede la locazione di memoria e quello che precede l'istruzione stanno ad indicare che si tratta di valori esadecimali. Le istruzioni sono, come si vede, memorizzate nelle locazioni di memoria a partire dalla 70 (esadecimale). Prima di eseguire il programma occorrerà "caricare" i due numeri (di un byte ciascuno) da sommare, nelle locazioni 80 e 81 (esadecimali). Ad esempio, per sommare i due numeri 11 e 23 (decimali) occorrerà porre:

&80 = 11
&81 = 23

e poi "eseguire" le istruzioni contenute nelle locazioni di memoria che iniziano alla 70.

Programmare un microcomputer o un microprocessore facendo uso delle istruzioni "di macchina", quali quelle ora esaminate, può risultare estremamente penoso. Un lieve aiuto viene offerto dalla possibilità di lavorare in Assembler. Questo è un linguaggio sostanzialmente coincidente con quello della macchina, ma dove i nomi dei codici operativi sono sostituiti con altri che hanno un immediato significato mnemonico e dove le locazioni di memoria contenenti gli operandi vengono indicate con delle etichette o LABEL, definite all'inizio del programma. Un opportuno COMPILATORE di uso generale (compilatore Assembler o "assemblatore") provvederà poi a convertire il programma scritto in linguaggio Assembler in "codice macchina" [2]. Ad esempio il programma esaminato, relativo alla somma di due numeri, in Assembler diviene:

.ADDIZIONE	label che definisce l'inizio della routine
LDA PRIMO	carica nell'accumulatore il numero contenuto nella locazione di memoria etichettata PRIMO
CLC	azzera il bit "carry"
ADC SECONDO	somma al contenuto dell'accumulatore il numero contenuto nella locazione di memoria etichettata SECONDO
STA SOMMA	memorizza il contenuto dell'accumulatore nella locazione di memoria etichettata SOMMA
RTS	fine del programma

Qui LDA sta per "Load Accumulator"; CLC per "Clear Carry"; ADC per "Add Accumulator"; STA per "Store Accumulator".

Una volta *assemblato* il programma occorre riempire le locazioni di memoria etichettati PRIMO e SECONDO con i due addendi, ed eseguire il programma.

Anche l'uso dell'Assembler risulta pesante per l'utente, dovendo questi far continuo ricorso ai registri, accumulatore, etc. In altre parole, anche la programmazione in Assembler è strettamente legata alla struttura del microprocessore, che l'utente deve sempre avere presente nel programmare un generico problema. Esistono in commercio numerosi *linguaggi* di livello più elevato, che liberano l'utente da tali vincoli. Tra questi, i più diffusi sono il C, il Basic, il Fortran. Altri come il Pascal, il Cobol, etc. stanno rapidamente diventando obsoleti. Negli ultimi anni hanno avuto una larghissima diffusione le versioni *grafiche* di alcuni di questi programmi: è il caso del C++, del VISUAL BASIC, etc.

Con la diffusione di Internet hanno acquistato grande importanza il Java ed il PERL. Non entreremo nei dettagli di alcuno di questi linguaggi, ma ci limiteremo ad

[2] Una limitazione del linguaggio Assembler è costituita dal fatto che ciascun tipo di microprocessore deve far uso di un proprio compilatore. Tale limitazione non è presente in linguaggi di livello più elevato, quali il C, il Fortran, il Basic etc.

illustrare come il programma precedente, relativo alla somma di due numeri, possa essere scritto in Fortran (vedasi la tabella 3.6).

```
      PROGRAM ADDIZIONE
      INTEGER PRIMO, SECONDO, SOMMA
      PRIMO=11
      SECONDO=23
      SOMMA=PRIMO+SECONDO
      PRINT 1001, SOMMA
1001  FORMAT(1x,I4)
      STOP
      END
```

Tabella 3.6: Esempio di routine in Fortran per effettuare la somma di due interi.

La prima riga definisce semplicemente il nome del programma. La seconda *dichiara* le variabili PRIMO, SECONDO e SOMMA come interi. La terza e quarta attribuiscono i valori ai due operandi (operazione che nell'esempio in Assembler era effettuata al di fuori del programma). La quinta istruzione calcola la somma e la successiva stampa il risultato in "formato" intero di quattro cifre, come specificato dalla settima istruzione, che ha una LABEL numerica 1001. Qui vediamo anche l'utilizzo di tali LABELS, che possono anche esser adoperate per effettuare dei "salti" (JUMP), attraverso istruzioni del tipo "GO TO 2000" (salta all'istruzione che ha la LABEL 2000).

Vediamo infine come il medesimo programma potrebbe esser scritto in linguaggio C (vedasi la tabella 3.7).

```
MAIN(){
INT PRIMO=11;
INT SECONDO=23;
INT SOMMA;
SOMMA=PRIMO+SECONDO;
PRINTF("%d \n", SOMMA);
}
```

Tabella 3.7: Esempio di routine in C per effettuare la somma di due interi.

Qui la prima riga dichiara il programma come un MAIN (per distinguerlo da una procedura *chiamabile* da altro programma) ed apre (parentesi {) la sequenza delle istruzioni, chiusa da un'analoga parentesi } all'ultima riga di codice. Le successive tre istruzioni attribuiscono i valori agli operandi e *dichiarano* questi ed il risultato come interi (INT). L'istruzione che segue calcola la somma e pone il risultato nella variabile SOMMA. Il risultato è poi stampato (PRINTF) come intero. Notiamo come la fine di ciascuna istruzione sia segnalata dalla punteggiatura ";".

Sia nel caso del programma scritto in FORTRAN che in quello in C, occorrerà effettuare una *compilazione* prima di poterlo eseguire. Il compilatore crea un

modulo, essenzialmente in linguaggio macchina, a partire dalle istruzioni scritte in FORTRAN on in C.

Gli esempi fatti, estremamente semplici, non rendono appieno l'idea del grado di semplificazione che si ottiene ricorrendo a tali linguaggi, rispetto all'utilizzo dell'Assembler. Un programma che esegua una gran quantità di calcoli numerici, e che in C comprenda qualche migliaio di istruzioni, in Assembler ne richiederebbe diverse decine di migliaia. Si deve necessariamente far ricorso all'Assembler solo in circostanze molto particolari; ad esempio quando si debba accedere in modo diretto a porte di I/O, o si debba programmare dei videogiochi. La credenza che utilizzando l'Assembler si guadagni in velocità di esecuzione và decisamente sfatata: i compilatori oggi disponibili, sia per il linguaggio C che per il FORTRAN, sono estremamente sofisticati ed è impresa difficilissima "batterli" programmando direttamente in Assembler.

3.4 BUS e standard di comunicazione

Nelle precedenti sezioni abbiamo fatto uso del concetto di BUS ed abbiamo accennato alla trasmissione di dati e/o programmi, sia tra componenti di un dato microcontrollore/microprocessore che con eventuali periferiche (stampanti, monitor, modem, dischi esterni, etc). Ci occuperemo ora di descrivere sia gli standard adoperati per la trasmissione che la struttura dei diversi tipi di BUS. Data l'enorme varietà di architetture disponibili oggi sul mercato e la complessità di queste, non potremo che fornire una sommaria descrizione delle principali caratteristiche di quegli standard che sono oggi maggiormente diffusi. Per i dettagli di particolari tipi di BUS e protocolli di comunicazione associati è necessario consultare le caratteristiche fornite dai produttori. Nel seguito di questa sezione discuteremo la struttura fisica ed i protocolli di comunicazione definiti da alcuni BUS standard.

Come visto in precedenza, un BUS è un canale di comunicazione condiviso da più dispositivi. Tali dispositivi dialogano tra loro secondo uno standard definito dalle specifiche del BUS stesso. Al livello fisico un BUS non è altro che un insieme di conduttori con il quale si interconnettono i vari sottosistemi. Tali connessioni elettriche vengono usate non soltanto al fine di scambiare dati tra di essi, ma anche, in alcuni casi, per ricevere alimentazione. Esistono numerosi BUS standard che si distinguono per numero di linee presenti, tipo di segnali elettrici utilizzati, lunghezza massima del BUS, presenza di un clock, velocità massima di trasferimento dati, modalità di trasmissione, topologia di connessione delle periferiche, etc. Un BUS possiede a livello logico tre linee di comunicazione:

- linee di controllo: utilizzate per il controllo del flusso dati: ad esempio per il trasporto dei segnali di inizio e fine delle transazioni o per gli interrupt request;
- linee di indirizzamento: che indicano a quale sottosistema la transazione si riferisce e l'indirizzo della cella di memoria interessata;
- linee dati: utilizzate per trasferire i dati.

I vantaggi di una simile architttura di comunicazione sono molteplici. Il sistema è estremamente versatile in quanto si possono facilmente aggiungere e rimuovere

periferiche; le schede di espansione possono quindi essere usate su qualunque sistema usi il BUS per il quale sono state costruite. In una tipica applicazione le linee fisiche del BUS sono condivise da molte periferiche e questo fà sì che risulti economico da realizzare. Naturalmente condividere le linee del BUS porta anche ad alcuni inconvenienti: la velocità di trasferimento dati è limitata dal "traffico" sul BUS quando troppe periferiche sono connesse contemporaneamente; la complessità del BUS stesso è maggiore, in quanto deve consentire la coesistenza di periferiche con tempi di risposta differenti, garantendo le comunicazioni tra essi. Esistono anche tipi di BUS in cui ogni periferica possiede la propria linea dedicata di comunicazione. Questi non presentano gli svantaggi appena visti ma sono in generale più complessi e "costosi" da implementare.

In sistemi dotati di più di una CPU, o comunque di più di un dispositivo che può iniziare transazioni sul BUS, occorre un meccanismo per stabilire quale dei dispositivi che chiedono l'accesso, debba averlo per primo. Tale meccanismo è noto come *arbitraggio*.

Un dispositivo che sia in grado di prendere il controllo del BUS è detto *BUS Master*, mentre uno che sia solo in grado di rispondere a transazioni iniziate da un Master è detto *BUS Slave*.

Non tutti i BUS permettono la presenza di più sottosistemi Master contemporaneamente. Se un BUS lo permette si dice che esso supporta il BUS mastering.

Figura 3.29: Esempio di un BUS con più dispositivi BUS Master presenti. Da "Computer Architecture: theme and variations" Alan Clements. Ringraziamo il Prof. A. Clements per il permesso di riprodurre questa figura.

Nella figura 3.29, la CPU è un BUS master, mentre la memoria di sistema è un BUS slave. Una delle porte di I/O è stata considerata *bus master*, poiché si è ipotizzato che anch'essa sia in grado di prender controllo del BUS (ad esempio, per trasferimento di interi blocchi di dati: DMA). L'altra periferica è un BUS slave, poiché essa è solo in grado di rispondere a richieste di lettura/scrittura.

La connessione tra il disco esterno ed il suo *controller* è anch'essa un BUS. La CPU è mostrata in un riquadro separato, con una propria memoria cui è collegata tramite un BUS locale.

Per il collegamento di periferiche (dischi veloci, videocamere, tastiere, etc.) si fa uso dei cosiddetti "BUS periferici" o BUS di I/O. Rientrano in questa categoria il BUS SCSI, FireWire, RS-232, USB, Ethernet, etc. In passato il collegamento di una periferica ad un computer risultava spesso penoso, dovendosi conoscere e settare in modo opportuno vari parametri (velocità di connessione, parità, bits di controllo etc.) sia nel computer che nella periferica. Oggi i dispositivi noti come *plug-and-play* risolvono il problema, poiché sono in grado di "negoziare" e settare per conto proprio i parametri necessari.

Modelli di trasferimento dati

Esistono differenti modi con cui un BUS può trasferire dati tra i differenti sottosistemi. Una comunicazione può essere *sincrona* o *asincrona*.

Nel primo caso una delle linee di controllo è una linea di clock e la comunicazione avviene secondo un protocollo di trasmissione che prevede che tutte le altre linee del BUS siano lette o impostate in coincidenza con il clock comune. Il vantaggio principale di una comunicazione sincrona è che è semplice da realizzare e richiede poca logica di controllo. Sull'altro versante il vincolo di un clock comune a tutte le periferiche è un limite all'utilizzo di tale tipo di BUS ed inoltre richiede, perché sia conservato il sincronismo, dei BUS relativamente corti. In una tipica comunicazione sincrona la comunicazione viene dapprima impostata tramite le linee di controllo e poi vengono trasmessi i dati ad ogni ciclo di clock.

Le comunicazioni asincrone invece non necessitano di un clock di riferimento e si possono adattare ad un maggior numero di periferiche. L'assenza di clock permette di avere linee di comunicazione piuttosto lunghe. Affinché una comunicazione asincrona sia possibile è però necessario un qualche protocollo di "handshaking"[3] e solitamente la lunghezza del messaggio trasmesso non può essere molto lungo o si incorre nel rischio della perdita di sincronia. Una tipica comunicazione asincrona inizia con una richiesta dati ad un sottosistema, richiesta che avviene sulle linee di controllo. Il sottosistema interessato risponde settando le linee dati ed impostando un'apposita linea di controllo per confermare che i dati sono presenti. Il sottosistema che aveva effettuato la richiesta legge i dati e comunica, sempre tramite le linee di controllo, che ha effettuato la lettura. A questo punto la comunicazione ha termine ed i sottosistemi rilasciano il BUS.

In generale, si tende ad usare BUS sincroni in applicazioni nelle quali si può far uso di periferiche diverse le cui velocità sono però molto simili. I bus sincroni tendono ad avere una minore latenza[4] e possono avere una maggiore larghezza di banda (bandwidth)[5].

[3]Per "handshaking" (letteralemente "stretta di mano") si intende lo scambio, tra un processore ed un dispositivo esterno, o tra due processori, delle informazioni necessarie perché la trasmissione di un pacchetto dati possa aver luogo. Un tipico segnale di "handshacking" può ad esempio essere costituito dall'invio, da parte del processore, di due bit predefiniti.

[4]La latenza di un BUS è il tempo richiesto per predisporre un trasferimento dati.

[5]La bandwidth è semplicemente il numero di bit che possono esser trasmessi al secondo. Ad esempio, un BUS che abbia una *larghezza* pari a 24 e lavori a 60 MHz, ha una bandwidth pari a

I BUS di uso "general purpose" tendono invece ad essere asincroni, poiché tale soluzione offre maggior flessibilità.

I BUS differiscono anche nel numero di linee con le quali i dati vengono trasmessi. Un BUS si dice *seriale* se i dati vengono trasferiti un bit alla volta su di un'unica linea; si dice invece *parallelo* se al contrario vengono trasferiti più bit ad ogni ciclo. Solitamente in un BUS parallelo il numero di bit trasferito è un multiplo di 8, cioè vengono trasferiti uno o più byte in parallelo.

Architetture di BUS

Tre differenti tipologie di BUS si sono sviluppate e si sono affermate nel tempo durante l'evoluzione dei personal computer e delle tecnologie elettroniche:

- i *backplane BUSes*;
- gli *I/O BUSes*;
- i BUS processore-memoria (detti anche front-side BUSes).

Questi ultimi sono BUS di connessione dedicati, ad alta velocità, con il solo compito di garantire un accesso diretto alla memoria centrale da parte del processore. Dipendono dal particolare tipo di microprocessore e di memoria presenti nel microcomputer e sono studiati in modo da garantire la massima efficienza di trasferimento dati tra di essi. Data l'alta velocità di trasferimento sono adatti al trasferimento di dati solo su corte distanze.

Un tipico backplane BUS consente di estendere le capacità di un microprocessore tramite apposite schede di espansione[6]. Esso è composto da un *backplane* dove il BUS risiede e nel quale sono presenti una serie di connettori, chiamati *slot*, montati perpendicolarmente ad esso. I backplane BUS sono per questo motivo estremamente flessibili nel numero e nel tipo di periferiche che possono supportare. Proprio per garantire questa varietà di periferiche supportate, e per permettere distanze di comunicazione maggiori, la velocità di trasferimento dati è inferiore a quella tipica della tipologia di BUS vista in precedenza. Uno dei vantaggi principali dei backplane BUS è che sono dei BUS standard e garantiscono quindi la compatibilità delle periferiche con tutti i sistemi aderenti a tale standard. Alcuni esempi di backplanes e di schede da inserirvi sono mostrati in figura 3.30 e 3.31.

L'ultimo tipo di BUS sono i BUS di I/O. Questi sono pensati per connettere dispositivi ad un microcomputer attraverso opportuni cavi di comunicazione. Caratteristica principale di questo tipo di BUS è la grande lunghezza che il BUS può raggiungere. Per questo motivo però la velocità di comunicazione è inferiore a quella dei precedenti tipi di BUS. Anche questi BUS, come quelli basati su backplane, consentono l'interfacciamento ad una grande varietà di periferiche e sono codificati in standard industriali.

Lo stato dell'arte dei moderni BUS di sistema usati nei PC e nell'industria è il risultato finale del susseguirsi di molte generazioni di tecnologie e del tentativo da parte dei costruttori di fornire uno standard comune. I primi microcomputer (si

$6\ 10^7 \times 24\ =\ 1.44\ Gb/s$.

[6]Una scheda di questo tipo può ad esempio essere costituita da un ADC, un modem, un'interfaccia per fotocamera, una memoria aggiuntiva etc.

Figura 3.30: Esempi di backplanes.

Figura 3.31: Scheda di espansione per un backplane VME.

pensi ad esempio all'IBM PC/XT basata sull'Intel 8088 con BUS ISA) possedevano un unico BUS di sistema al quale erano connessi tutti i suoi sottosistemi: microprocessore, memoria, dischi rigidi, schede di espansione e periferiche di I/O. Tale approccio garantiva la semplicità dell'architettura ed il suo basso costo si adattava bene alle necessità dell'epoca. Con l'avanzare delle tecnologie fu evidente però che il collo di bottiglia principale era proprio la lentezza di un tale BUS. Quando le nuove CPU, quali l'intel 80486, capaci di lavorare a frequenze di clock elevate, divennero popolari, fu chiaro che bisognava trovare un modo per ovviare a tale inconveniente. Se da un lato, infatti, tali BUS erano sufficienti per le comunicazioni tra le perifieriche più lente, essi non lo erano certamente per quei dispositivi che necessitavano di trasferire grosse quantità di dati velocemente (si pensi ad esempio alle schede video o al microprocessore stesso). La soluzione fu l'introduzione delle architetture *local BUS*. Queste prevedono una struttura a tre livelli di BUS. Il primo livello è costituito dal front-side BUS con il quale si garantisce al microprocessore l'accesso dedicato alla memoria. Il secondo livello è costituito da un backplane BUS e connette le periferiche "veloci", ovvero quei dispositivi che necessitano di grande larghezza di banda. Tutte le altre periferiche sono connesse sul terzo livello di BUS, il BUS di I/O a bassa velocità. Il collegamento tra i vari livelli di BUS è fornito da appositi chip che fungono da *bridge*, consentendo così anche ai dispositivi non

Figura 3.32: Differenti architetture dei BUS. Da "Computer Architecture: theme and variations" Alan Clements. Ringraziamo il Prof. A. Clements per il permesso di riprodurre questa figura.

presenti sul front-side BUS il dialogo con il microprocessore e la memoria centrale. Questo approccio elimina il collo di bottiglia presente quando tutte le periferiche devono risiedere sullo stesso BUS. Schemi dei due tipi di BUS descritti in questo paragrafo sono mostrati in figura 3.32. Quella in alto mostra un sistema in cui un unico BUS è condiviso da tutte le periferiche che possono essere BUS Master o Slave; la seconda rappresenta invece un'architettura schematizzata a tre livelli di BUS.

3.4.1 Backplane BUSes

ISA

Il BUS ISA (*I*ndustry *S*tandard *A*rchitecture) fu introdotto da IBM come parte del suo rivoluzionario PC/XT nel 1981. Il primo ISA XT era sotto il completo controllo del microprocessore (l'Intel 8088) e il numero di linee di indirizzo, di dati e la velocità del BUS furono limitate a quelle del processore.

Il BUS XT ha un design relativamente semplice e possiede 53 linee di segnale ed 8 linee tra alimentazione e terra. Esso è un BUS *sincrono*, che cioè opera secondo

la temporizzazione fornita da un clock di sistema uguale per tutte le periferiche connesse ad esso; è un BUS a 8 bit, cioè in grado di trasferire 8 bit di dati alla volta, con controllo di parità; le linee di interrupt sono edge-triggered, ovvero gli interrupt sono segnalati dal cambio di tensione sulla linea: questo implica che le linee di interrupt non possano essere condivise da più periferiche. Il BUS XT non supporta il BUS mastering delle periferiche e quindi gli unici dispositivi che potevano assumere il controllo del BUS nell'IBM PC/XT erano il microprocessore ed il controller DMA. Le caratteristiche principali dell'ISA XT sono:

- 20 linee di indirizzamento;
- 8 linee di dati;
- 6 linee di interrupt request;
- 3 linee DMA;
- velocità di clock: 4.77 MHz;
- velocità di trasferimento dati: 4.77 MB/s.

Il BUS XT fu caratterizzato dalla semplicità del suo design e, sebbene non introdusse alcuna caratteristica particolarmente innovativa, era ben adatto alla particolare architettura per la quale era nato.

Proprio la semplicità del BUS XT ne limitò le potenzialità e, per far fronte alle nuove esigenze, una versione evoluta del BUS fu introdotta nel 1984 da IBM con il BUS ISA AT (*A*dvanced *T*echnology). Una caratteristica importante dell'ISA AT fu la sua compatibilità con il precedente BUS XT. I principali miglioramenti dell'ISA AT rispetto al suo predecessore furono l'aumento del numero linee dati (16) e di indirizzamento (24) e della velocità di clock (a 8 MHz).

PCI

Nel 1992 Intel annunciò la propria tecnologia local BUS, denominata PCI (*P*eripheral *C*omponent *I*nterconnect). Quello che Intel voleva era un BUS generico adattabile a tutte le esigenze. L'obiettivo era quello di ottenere un BUS veloce ed economico da realizzare. Inoltre si voleva rendere tale BUS indipendente dal tipo di microprocessore utilizzato. La soluzione di Intel fu quella di isolare completamente il BUS locale del microprocessore dal resto a cui il BUS PCI si interfaccia tramite un bridge (figura 3.33). Nonostante ciò il PCI è un BUS locale in quanto, come nel caso del VLB, ad esso si connettono direttamente le periferiche "veloci" mentre quelle "lente" possono essere disposte su un altro BUS collegato al PCI tramite l'apposito bridge. Rendere completamente indipendente il BUS del microprocessore da quello di tutte le altre periferiche si è dimostrata un'idea vincente in quanto, da un lato non pone limiti alla possibilità di sviluppo dei processori che possono avere BUS locali veloci studiati per le proprie esigenze, dall'altra, rendendo indipendenti anche le periferiche, garantisce la compatibilità di queste con le generazioni di tecnologie future. Non a caso, infatti, PCI è tuttora lo standard dominante nel mercato dei personal computer. Era chiaro dunque che il modo migliore per risolvere i problemi di larghezza di banda dati delle periferiche era di dividerle secondo le loro necessità

Figura 3.33: Architettura del BUS PCI. Da "Computer Architecture: theme and variations" Alan Clements. Ringraziamo il Prof. A. Clements per il permesso di riprodurre questa figura.

e costruire una gerarchia di BUS locali indipendenti tra loro e comunicanti tramite appositi bridge. Questa soluzione garantisce una flessibilità enorme ed un'elevata adattabilita alle nuove tecnologie, mantenendo la compatibilità con gli standard preesistenti: è per questo che i moderni sistemi (in primo luogo, ma non solo, i personal computer) hanno adottato tale architettura.

Le caratteristiche tecniche del BUS PCI sono:

- BUS sincrono a 32 o 64 bit di dati e indirizzi;
- possibilità di operare con logica a 5 o 3.3 Volt;
- velocità di clock a 33 o 66 MHz;
- velocità di trasmissione dati fino a 132 MB/s (32 bit mode) o 264 MB/s (64 bit mode) a 33 MHz;
- piena compatibilità con il BUS mastering;
- supporto completo per il DMA;
- configurazione automatica delle periferiche (PNP).

La grande versatilità del BUS PCI ne determinò il successo, che dura ancor oggi. Nel corso degli anni il BUS PCI ha dato luogo a numerosi standard industriali (ad esempio Compact PCI, Mini PCI, PC104-Plus, PISA, P2CI, PMC, IPCI, CardBUS) e recentemente è nata un sua un'estensione denominata PCI-X. Quest'ultima, compatibile con lo standard originale, si differenzia per una maggiore velocità di clock che può arrivare fino a 533 MHz consentendo così un trasferimento dati fino a 4.3 GB/s. PCI-X non sembra comunque destinato al successo del predecessore, a causa dell'avvento di una nuova tipologia di BUS (vedi PCIe nel seguito).

AGP

A fianco del BUS PCI sono presenti in un normale personal computer altri tipi di BUS per soddisfare diverse esigenze. Nel seguito vedremo alcuni di questi BUS, ad esempio quelli dedicati alle periferiche "esterne". Tra i BUS specifici che più comunemente vengono utilizzati oggi vi è il BUS AGP (*A*ccelerated *G*raphics *P*ort). Come il nome suggerisce tale BUS è nato per venire incontro alle necessità dei moderni acceleratori grafici. Lo standard fu presentato da Intel nel 1997 nella sua versione 1.0, ed è oggi arrivato alla versione 3. AGP è un'estensione di PCI pensato come BUS locale per schede video. Esso definisce 20 nuove linee di segnale, un nuovo connettore, tensioni di lavoro di 1.5 o 3.3 V. La versione 3.0 dello standard definisce velocità di clock multiple di 66 MHz nelle varianti 1x, 2x, 4x e 8x, arrivando in tal modo ad una frequenza massima di 533 MHz. La larghezza del BUS dati è di 32 bit; pertanto la velocità di trasferimento dati massima è di 2.1 GB/s. AGP, come PCI sul quale è basato, è un BUS sincrono e possiede quindi un clock comune di riferimento; ma possiede anche modalità di trasferimanto dati asincrona. AGP è un esempio di BUS locale dedicato ad una particolare tipologia di periferica. Un altro esempio di tali BUS sarà mostrato nel seguito con i BUS dedicati alle memoria di massa.

PCI Express

Lo standard PCIe (*PCI E*xpress) definisce un tipo di BUS di nuova generazione. Mentre nei BUS visti finora, le periferiche condividono il BUS fisico che trasporta comandi, dati e indirizzi, nel BUS PCIe le linee hardware non vengono condivise ma le periferiche sono connesse ad un livello logico superiore. Infatti una delle caratteristiche peculiari di PCIe è la differente topologia con la quale le periferiche vengono connesse sul BUS PCIe. Non condividendo lo stesso BUS fisico, le periferiche PCIe sono connesse individualmente ad uno *switch* dedicato che "smista" le comunicazioni da e verso le periferiche ad esso connesso, consentendo in tal modo le comunicazioni tra esse. Un'altra differenza primaria rispetto ai BUS finora visti è il fatto che PCIe è un BUS seriale anziché parallelo (come è ad esempio PCI). Questo significa che il BUS dati ha una larghezza pari ad 1 bit. PCIe trasferisce sia dati che comandi in modo asincrono usando un BUS con 4 sole linee, denominate PCIe *link*, formato da una coppia di linee per i segnali in ingresso ed una per quelli in uscita. Al livello fisico i segnali utilizzati da PCIe sono differenziali di tipo LVDS (Low Voltage Differential Signaling) trasmessi alla velocità di 2.5 Gb/s in entrambe le direzioni indipendentemente. PCIe è pensato per essere altamente scalabile: è possibile "unire" più link per aumentare la banda dati; lo standard prevede in tal modo velocità di trasferimento che vanno da 2.5 Gb/s per un link 1x fino a 80 Gb/s per un link 32x ottenuto combinando 32 link 1x.

CAMAC

Il CAMAC (*C*omputer *A*utomated *M*easurement *A*nd *C*ontrol) è un sistema modulare nato nel 1969 dalla necessità di uno standard per le periferiche di presa e gestione dati negli esperimenti di fisica nucleare. Un sistema CAMAC è composto da strumenti modulari inseriti in un "crate" (tipicamente con 25 alloggiamenti det-

ti stazioni). I crate ospitano il *dataway* al quale i moduli si interfacciano tramite un connettore a 86 pin. Il dataway fornisce le tensioni di alimentazione ai moduli, nonché il BUS di indirizzamento, il BUS dati e quello di controllo. Le stazioni numero 24 e 25 ospitano un modulo speciale detto *crate controller* che ha il compito di gestire le comunicazioni tra i moduli e il computer host al quale sono connessi. Durante un'operazione, il controller genera sul dataway un comando che consiste in un segnale sulla linea della stazione alla quale è destinata la richiesta, il numero a 4 bit della sottostazione sulle linee di sottoindirizzamento, 5 bit indicanti la funzione richiesta sulle linee di funzione ed infine imposta la linea di *Busy* al livello logico 1 ad indicare che è in corso un'operazione. Il modulo richiesto genera una risposta di comando accettato e agisce eseguendo la funzione richiesta. Se il comando richiede trasferimento dati vengono usate le linee di lettura o scrittura.

VME

Il BUS VME (*V*ersa *M*odule *E*uropa) è stato introdotto nel 1981 per far fronte alla necessità di avere un BUS flessibile e potente e non legato alla particolare architettura di un microprocessore e permetterne in tal modo la coesistenza su di un unico BUS. Sono previste larghezze di BUS dati e indirizzi selezionabili dinamicamente per far fronte all'eterogeneità delle architetture esistenti: sono possibili indirizzamenti a 16, 24 o 32 bit e trasferimento dati di 8, 16 o 32 bit per ciclo. La velocità massima di trasferimento dati è di 40 MB/s. Ogni device presente sul BUS possiede un indirizzo o un insieme di indirizzi. È possibile il DMA e VME supporta il BUS mastering. Al contrario della maggior parte dei BUS incontrati in precedenza il trasferimento dati avviene in modo asincrono.

3.4.2 I/O BUSes

I BUS periferici permettono di connettere periferiche quali dischi rigidi, cd-roms, stampanti, scanners, ecc. ad un microcomputer. I BUS di I/O appartengono al secondo dei livelli di BUS delineati in precedenza e sono pilotati da un controller che generalmente risiede sul BUS locale. Nel seguito vedremo alcuni degli standard più diffusi.

ATA

L'ATA (*A*dvanced *T*echnology *A*ttachment) fu definito come standard all'inizio degli anni 90 e da allora ha subito una continua serie di evoluzioni per tenersi al passo con le tecnologie delle memorie di massa (specialmente dischi rigidi) sempre più veloci e sofisticate. Ancora oggi ATA è lo standard più diffuso tra gli hard disk (questa popolarità è in gran parte dovuta al fatto che le periferiche ATA sono più economiche di quelle che fanno uso di tecnologie concorrenti). La varietà di nomi connessa a questo standard è incredibile e fuorviante nel momento in cui vengono usati anche termini impropri (quali IDE) per riferirsi ad esso. Gli standard, noti con i nomi di ATA-1 fino ad ATA-7, definiscono modalità di trasferimento dati programmato, i cosidetti PIO modes, e modalità di accesso diretto alla memoria, DMA modes, introdotti nel momento in cui la velocità di trasferimento è divenuta una necessità. I PIO modes e i DMA modes differiscono per la velocità di trasferimento e le modalità

con le quali i dati vengono traferiti. Il primo standard noto come ATA-1 prevedeva una velocità di trasferimento dati massima di 8.3 MB/s e definiva i PIO modes 0 e 1 mentre non era previsto alcuna modalità di trasferimento DMA; lo standard ATA-7 (anche detto Ultra-ATA/133) arriva a trasferire 133 MB/s e prevede le modalità PIO modes 0,1,2,3,4 e DMA modes 0,1,2 e Ultra-DMA modes 0,1,2,3,4 e 5. Come si vede è presente una grande varietà all'interno di questo standard in continua evoluzione. Tutti gli standard ATA garantiscono la compatibilità con le precedenti modalità di trasferimento, per cui, ad esempio, un controller ATA-7 può gestire periferiche ATA-5. Una delle poche, se non l'unica, caratteristica rimasta invariata nel tempo è il numero di linee del BUS che è 40 con una larghezza del BUS dati di 8 bit per le versioni più vecchie e di 16 bit per le più recenti. Uno dei limiti dei BUS ATA è il limitato numero di periferiche che possono essere connesse contemporaneamente. Esso era di sole due periferche negli standard più vecchi, ed è attualmente di quattro.

SCSI

Lo standard SCSI (*S*mall *C*omputer *S*ystems *I*nterface) definito nel 1986 è probabilmente lo standard più vecchio ancor oggi in uso nei moderni sistemi. SCSI fu pensato per essere facilmente espandibile, veloce e dotato di una interfaccia di alto livello. Naturalmente questi vantaggi rispetto alle interfaccie ATA, portarono ad una notevole complessità delle periferiche SCSI e ad un conseguente maggior costo delle stesse. Come nel caso di ATA, anche SCSI si è evoluto nel corso degli anni e sono stai definiti vari standard noti con i nomi da SCSI-1 a SCSI-4. Il numero delle periferiche connesse era di 7 per SCSI-1 e arriva fino a 16 per le specifiche SCSI-3 e SCSI-4. La larghezza del BUS dati è 8 o 16 bit. Il BUS supporta sia modalità di trasferimento sincrono che asincrono. La massima velocità di trasferimento dati è di 5 MB/s per SCSI-1 ed arriva fino a 320 MB/s secondo le specifiche SCSI-4.

RS-232 – standard per comunicazioni seriali sui PC

RS-232 fu originariamente approvato dalla Electronic Industries Association (EIA) nel 1960 e si modificò in seguito nel corso degli anni fino alla terza versione RS-232C del 1969 usata ancor oggi dalla maggior parte dei personal computer. Lo standard non specifica un tipo di connettore, e si sono diffusi due modelli: il DB25 a 25 pin ed il più comune DB9 a 9 pin (vedi figura 3.34). La trasmissione è asincrona. In questo tipo di trasmissione, un bit di START ed un bit di STOP vengono aggiunti all'inizio ed alla fine di ciascun gruppo di 8 bit. A volte i bit di STOP sono due. La velocità di trasmissione è misurata in BAUD (cioè in cambiamenti di stato/secondo). Valori comunemente adoperati sono: 300, 1200, 2400, 4800, 9600, 19200 e 38400 BAUD. Nello standard di trasmissione RS-232C lo zero logico è una tensione superiore a +3V (più precisamente nell'intervallo $+5V \div +15V$); l'uno logico una tensione inferiore a -3V (più precisamente nell'intervallo $-5V \div -15V$). La linea viene normalmente tenuta al valore logico uno. La trasmissione di una parola ha inizio quando la linea viene portata allo zero logico (START bit). Segue la stringa di bit che costituiscono il carattere (con il bit meno significativo inviato per primo) ed infine lo STOP bit.

All'estremità di ricezione, una porta UART (Universal Synchronous/Asynchronous Receiver/Transmitter) viene attivata dallo START bit. Essa trasferisce gli otto bit che seguono su di un BUS parallelo e viene poi resettata dallo STOP bit.

Figura 3.34: Connettori DB9 (sinistra) e DB25 (destra), maschi (sopra) e femmine (sotto).

Lo standard RS-232C prevede che, nella comunicazione tra due dispositivi A e B generici, una linea (oltre alla relativa linea di massa) venga adoperata da A per inviare dati a B ed una venga adoperata da B per inviare dati ad A. Se A è un computer o terminale (T) e B una periferica (C) saranno quindi necessarie almeno tre linee: una che connette la porta UART di *scrittura* di T alla porta di *lettura* di C, una seconda che connette la porta UART di *lettura* di T con quella di *scrittura* di C, ed una linea di massa. In realtà, lo standard prevede anche delle ulteriori linee, dette di *handshaking*, che possono essere utilizzate da ciascuno dei due dispositivi per comunicare all'altro che i dati sono pronti ad esser letti o che nuovi dati sono richiesti. In particolare si hanno altre quattro linee, due delle quali corrispondono a richieste che il dispositivo T (chiamato Data Terminal Equipment o DTE) invia al dispositivo C (noto come Data Communication Equipment o DCE) mentre le altre due corrispondono a richieste che il DCE invia al DTE.

Le prime due di queste quattro linee sono note come RTS (Request To Send) e DTR (o Data Terminal Ready). Le altre due sono note come CTS (Clear to Send) e DSR (Data Set Ready).

Alcuni dispositivi oggi in uso evitano l'uso di queste linee di *handshaking* ma ricorrono a quello che è noto come *software handshaking*. Per una discussione approfondita di questi punti si rinvia alla letteratura specializzata.

IEEE-1284 – porta per comunicazioni parallele sui PC

IEEE-1284 è uno standard che definisce comunicazioni parallele bidirezionali. Tali porte di comunicazione sono diffuse sui personal computer soprattutto per l'interfacciamento a stampanti. Lo standard IEEE-1284 definito nel 1994, ha rimpiazzato lo standard Centronix ormai obsoleto pur mantenedo la compatibilità con esso. Il nuovo standard prevede l'utilizzo di un connettore DB25 con 17 linee di segnale e 8 di terra. Le linee di segnale sono divise in:

- 4 linee di controllo;
- 5 linee di stato;

- 8 linee di dati.

Le linee di controllo sono usate come interfaccia di controllo e trasportano i segnali di handshaking tra i due device connessi. Le linee di stato sono usate in parte nel protocollo di handshaking e come indicatori di stato per cose quali fine carta, indicatore di linea occupata ed errori di interfaccia o della periferica.

Lo standard IEEE-1284 prevede 5 modalità di trasferimento dati:

- *Compatibility Mode*: anche detto Centronix o standard mode, è il modo di trasmissione dati del vecchio standanrd. Prevede che il trasferimento dati sia unidirezionale dal pc alla periferica e che vengano trasferiti 8 bit di dati alla volta sulle linee dati. La velocità massima di trasmissione è di circa 150 kB/s;

- *Nibble Mode*: questo modo permette di trasferire dati dalla periferica al pc ed è quindi spesso usato assieme al Compatibility Mode. Prevede il trasferimento di 4 bit di dati alla volta usando le linee di stato. La velocità di trasferimento dati è di circa 50 kB/s;

- *Byte Mode*: prevede il trasferimento dei dati 8 bit alla volta sulle linee dati dalla periferica al pc;

- *EPP Mode*: (Enhanced Parallel Port) fu sviluppato come standard di trasferimento dati ad alta velocità. Esso era in grado di trasferire 8 bit di dati (utilizzando le linee dati) in modo sincrono con il BUS ISA in entrambe le direzioni (da e verso la periferica). La velocità di trasferimento dati è di circa 1.5 MB/s;

- *ECP Mode*: (Extended Capability Port) è un modo avanzato di trasferimento dato che aumenta le prestazioni del modo EPP portando la velocità di trasferimento dati a circa 2.5 MB/s.

USB

Lo standard USB (*U*niversal *S*erial *B*us) costituisce un'interfaccia per il collegamento ai PC di periferiche quali stampanti, scanners, dischi, etc. (come nel caso di RS-232) ma anche di dispositivi di misura, controllo e simili, che stanno acquistando una grandissima diffusione.

USB è inoltre un BUS seriale estendibile: ad una porta USB è possibile collegare uno *hub* (ovvero un'estensione di porta) dotato di più porte analoghe, a ciascuna delle quali può essere attaccato un ulteriore *hub* e via dicendo. In totale è possibile attaccare ad una porta USB fino ad un massimo di 127 dispositivi diversi. I cavi di collegamento contengono due conduttori per i segnali, un conduttore di massa ed un conduttore di alimentazione (+5 V). In tal modo è possibile alimentare periferiche che siano sprovviste di una propria alimentazione (purché esse assorbano complessivamente meno di 100 mA). La figura 3.35 mostra, nella parte sinistra, una sezione del cavo. Nella parte destra sono mostrati alcuni tipici connettori.

I segnali sulle linee sono differenziali, con livelli di tensione di 0V e 3.3 V.

Esistono tre versioni, compatibili tra loro, dello standard USB. Le caratteristiche di queste sono mostrate nella tabella 3.8.

Specifica	Velocità pratica di trasmissione	Applicazione tipica	Caratteristica
"Low speed" USB 1.1 (1.5 Mb/s)	Bassa (100-100 kb/s)	Tastiera, Mouse ...	Costo minimo
"Full Speed" USB 1.1 (12 Mb/s)	Media (0.5-10 Mb/s)	Stampanti, Scanner ...	Larghezza di banda garantita
"High Speed" USB 2.0 (500 Mb/s)	Alta (25-500 Mb/s)	Video, Dischi esterni..	Alta Larghezza di banda garantita

Tabella 3.8: Specifiche USB e velocità di trasmissione.

Come si vede, la più recente (USB 2.0) è anche la più veloce, con una velocità massima di 500 Mb/s.

La comunicazione tra un computer (host) ed una periferica (device) è basata su di un complesso protocollo ed è organizzata in pacchetti (packets).

I tipi di pacchetti rilevanti per un USB a bassa velocità sono mostrati nella tabella 3.9.

In aggiunta ai pacchetti mostrati, esistono delle condizioni aggiuntive quasi statiche (reset, sospendi, ri-inizia).

Una transazione USB è un insieme di pacchetti scambiati tra l'*host* e la periferica (device). Una transazione ha sempre inizio con l'*host* che invia un TOKEN (SETUP, IN, OUT) al device. I device non sono in genere in grado di iniziare transazioni. Il pacchetto TOKEN contiene l'indirizzo del device nonché l'ENDPOINT all'interno di esso. l'ENDPOINT specifica il conduttore (IN o OUT) collegato.

Ciascun pacchetto è composto da diversi campi. Questi sono:

- Sync

  Ogni pacchetto deve iniziare con questo campo, che è una stringa di 8 bit nelle due modalità di trasmissione a velocità più bassa, mentre è di 32 bit in quella a velocità più elevata. Tale campo viene adoperato per sincronizzare il clock

Figura 3.35: Bus USB. A sinistra è mostrata una sezione del cavo; a destra, alcuni connettori.

Nome	Gruppo	Funzione
SETUP	TOKEN	Inizia un trasferimento di controllo
IN	TOKEN	Inizia un trasferimento dati verso l'host
OUT	TOKEN	Inizia un trasferimento dati al device
DATA0	DATA	Trasferisci da 0 ad 8 bytes di dati
DATA1	DATA	Trasferisci da 0 ad 8 bytes di dati
ACK	HANDSHAKE	Informazione accettata
NAK	HANDSHAKE	Occupato; invia successivamente
STALL	HANDSHAKE	Informazione non corretta

Tabella 3.9: Tipi di pacchetti rilevanti per un USB a bassa velocità.

del ricevitore con quello del trasmettitore. Gli ultimi due bit indicano il punto di partenza del campo successivo che è il PID.

- PID

  questo sta per Packet ID ed è quello che indica il tipo di pacchetto che viene inviato. Ad esempio, PID=1101 sta per SETUP, PID=0010 sta per ACK Handshake, PID=0011 per DATA0 etc.. Come si vede il PID ha solo 4 bit; tuttavia, per controllare che il PID venga ricevuto correttamente, i 4 bit vengono complementati ed aggiunti ai primi quattro, con che il PID diviene una stringa di 8 bit. Ad esempio un PID di SETUP diviene: 11010010

- ADDR

  questo è l'indirizzo del dispositivo cui il pacchetto è destinato. Ad esso sono riservati 7 bit, il che implica un massimo di 127 dispositivi. L'indirizzo 0 è non assegnato. Un dispositivo a cui non sia ancora stato assegnato un indirizzo dovrà rispondere ai pacchetti inviati all'indirizzo 0.

- ENDP

  è il campo di endpoint (4 bit)

- CRC

  è un campo di controllo (Cyclic Redundancy Check)

- EOP

  End of Packet

Riassumendo, vediamo che l'inizio della transazione avviene quando l'host invia al device un TOKEN, ad esempio OUT[7], costituito da un pacchetto contenente i campi:

Sync PID ADDR ENDP CRC EOP

[7]eventualmente preceduto da un pacchetto SETUP

Questo pacchetto specifica al device connesso all'indirizzo ADDR (endpoint ENDP) che l'host è pronto ad inviare dati (IN ed OUT sono sempre riferiti all'host).

Successivamente l'host invia un pacchetto di dati. Tale pacchetto sarà composto dai seguenti campi:

Sync PID DATA CRC EOP

Dove DATA può avere una lunghezza massima di:

- 8 bytes per dispositivi a bassa velocità
- 1023 bytes per dispositivi a piena velocità
- 1024 bytes per dispositivi ad alta velocità

Una volta ricevuto il pacchetto dei dati il device risponderà con un pacchetto di handshake, per segnalare che i dati sono stati correttamente ricevuti. Tale pacchetto è costituito dai campi:

Sync PID EOP

Se il device non è subito pronto, esso invierà un pacchetto NAK, per chiedere all'host di inviare il pacchetto in un secondo momento.

3.5 Il PIC®16F877

3.5.1 Introduzione

Nelle sezioni precedenti abbiamo fornito una descrizione della struttura e dell'utilizzo di un microcontrollore. In questa sezione ci proponiamo di fornire una descrizione, alquanto più dettagliata, di un microcontrollore particolare, il PIC®16F877 della Microchip®. Il circuito integrato PIC®16F877 [12], fa parte della famiglia PIC®16F87X, composta da quattro microcontrollori a memoria FLASH (la lettera F nel nome indica proprio la presenza della memoria di tipo FLASH). Tali microcontrollori sono prodotti dalla ditta *Microchip®*. La sigla PIC®, utilizzata per la prima volta dalla ditta *General Instruments*, è l'abbreviazione di "Programmable Intelligent Computer". Attualmente tuttavia si tende ad interpretare questa sigla come "Peripheral Interface Controller".

I microcontrollori che appartengono a questa famiglia offrono funzionalitá molto simili, differenziandosi tra loro essenzialmente per la dimensione delle varie memorie e il numero di pin di input/output a disposizione. Essendo piuttosto facili da programmare, essi sono stati utilizzati in una grande varietá di applicazioni.

3.5.2 L'architettura del PIC®16F877

Il microcontrollore è composto dal microprocessore, vero e proprio cuore del sistema, che si occupa di eseguire le operazioni matematiche e di gestire l'esecuzione dei programmi, e da vari dispositivi interni dedicati a compiti specifici, detti periferiche

Figura 3.36: Fotografia dell'integrato della Microchip® PIC®16F877.

interne o moduli. Una caratteristica peculiare dei PIC®16F87X è l'adozione di un'architettura di tipo "Harvard", che permette alla CPU di gestire istruzioni e dati attraverso memorie e bus separati. Questo tipo di architettura permette alla CPU di effettuare contemporaneamente un accesso alla memoria programmi ed uno alla memoria dati. La velocità di esecuzione di un'istruzione risulta quindi maggiore rispetto a quella ottenuta con un'architettura tradizionale (detta di "Von Neuman"), nella quale dati e programmi vengono prelevati attraverso il medesimo bus.

Il microcontrollore è realizzato in un circuito integrato contenuto in un involucro detto DIL40 che presenta 40 pin, 20 da ciascun lato (vedi figura 3.36). I pin sono virtualmente numerati da 1 a 40 come mostrato in figura 3.37.

Come per tutti i circuiti integrati ognuno dei pin è utilizzato per una o più funzioni, indicate da una apposita sigla. L'architettura generale di un microcontrollore ed il suo funzionamento sono stati discussi nella sezione 3.3. In questa sezione vengono descritte le caratteristiche specifiche del microcontrollore PIC®16F877.

L'architettura del PIC®16F877 è schematizzata nel diagramma a blocchi mostrato in figura 3.38. In questa figura sono riportati tutti i componenti che saranno descritti nella prossime sezioni.

3.5.3 Pin per le funzioni di base

Iniziamo la descrizione delle funzioni associate ai vari pin, a partire da quelli che assicurano le funzioni basilari del microcontrollore: alimentazione, clock ed inizializzazione. Questi pin devono essere sempre connessi.

- Alimentazione: la tensione di riferimento per il livello logico 0 (0 V) è collegata ai pin 12 e 31, indicati con V_{ss}, mentre i pin 11 e 32, indicati con V_{dd} vengono collegati alla tensione di riferimento per il livello logico 1 (5 V).

- Clock: come per tutti i microprocessori è necessario avere un segnale di riferimento per mantenere la sincronia nell'esecuzione delle operazioni. Nel caso del 16F877 è possibile derivare il segnale di clock da circuiti costituiti da cristalli o da circuiti oscillanti a feedback basati su resistenze, condensatore ed elementi attivi. Il clock può avere varie frequenze di oscillazione fino ad un massimo

Figura 3.37: Schema del circuito integrato PIC®16F877. Accanto ad ogni piedino è riportata la numerazione e la sigla che indica le funzioni ad esso associate.

di 20 MHz. Usualmente si realizza il circuito di clock utilizzando un quarzo che permette di ottenere una elevata stabilità sia temporale che al variare della temperatura ed alla stesso tempo offre una frequenza di oscillazione molto precisa. Il segnale generato dal quarzo viene connesso ai pin 13 e 14 (OSC1 ed OSC2) utilizzando il circuito mostrato in figura 3.39.

Il tempo che intercorre tra due impulsi di clock successivi è detto ciclo del clock. Il PIC®16F877 esegue un'istruzione elementare, cioè un ciclo macchina, ogni quattro cicli di clock e quindi, nell'ipotesi di utilizzare la massima frequenza di clock, può eseguire un'istruzione elementare ogni 200 ns. Questo microprocessore è quindi capace di eseguire 5 milioni di operazioni elementari al secondo. Tale numero, apparentemente molto grande, è in effetti piccolo rispetto alle prestazioni di un PC attuale, che funziona ad almeno 2 GHz corrispondente a 2 miliardi di cicli di clock al secondo.

- Inizializzazione: il pin 1 denominato $MCLR$ è utilizzato per inizializzare il microprocessore. Se questo pin viene connesso a massa il microprocessore arresta l'esecuzione del programma ed esegue la routine di reset. Questa routine usualmente carica in tutti i registri i valori iniziali ed inizia ad eseguire il programma in memoria a partire dalla prima istruzione. Affinché il microprocessore funzioni correttamente questo pin deve quindi essere collegato a 5 V. Esso è spesso collegato ad un pulsante di RESET che permette di mettere a massa temporaneamente il pin 1 provocando una riinizializzazione del microcontrollore.

Figura 3.38: Diagramma a blocchi dell'architettura del microcontrollore PIC®16F877 [12].

3.5.4 Le porte del PIC®16F877

Con l'eccezione dei pin che assicurano le funzioni di base discusse nel paragrafo precedente, tutti gli altri pin del microprocessore corrispondono ad ingressi o uscite digitali. Molti di essi hanno anche altri ruoli ma questi saranno discussi più avanti. Tutti i pin di ingresso ed uscita del PIC16F877 sono suddivisi in gruppi detti *porte*. In totale si hanno 5 porte indicate oguna con una lettera: A, B, C, D ed E. La porta A comprende 6 pin, le porte B, C e D ne comprendono 8 mentre la porta E ne comprende 3. I pin appartenenti ad ogni porta sono numerati da 0 ad n e quindi ogni

Figura 3.39: Schema del circuito di clock del PIC®16F877 realizzato con un cristallo [12].

pin viene indicato con una sigla che indica la porta ed il numero del pin in quella porta. Ad esempio *RA*3 indica il pin 4 (la numerazione parte da 0) della porta A. È possibile individuare in figura 3.37 i pin appartenenti alle varie porte ed in figura 3.38 la loro posizione nello schema a blocchi. Le sigle indicate a fianco del nome del pin delle varie porte indicano le eventuali altre funzioni associate al pin. In totale sono disponibili 33 pin di input/output. Il grande numero di pin disponibili per ingressi ed uscite sono uno dei punti di forza di questo particolare microcontrollore. In effetti essi non sono sempre tutti disponibili poiché alcuni possono essere utilizzati per funzioni particolari associate ai moduli interni del microprocessore oppure possono essere dedicati a connessioni con periferiche esterne, come ad esempio lo schermo o la tastiera

Tutti i pin delle porte sono configurabili come ingressi o uscite del microprocessore. La struttura generale delle 5 porte è grosso modo la stessa. Ogni porta è configurata utilizzando il registro di direzione dei dati. Questo fa parte dei registri di stato discussi nella sezione 3.3.4 di questo capitolo. Esso è indicato come TRIS seguito dalla lettera che individua la porta. Ad esempio TRISA è il registro di direzione dei dati che corrisponde alla porta A. Ogni bit di questo registro corrisponde ad un pin della porta, il bit 0 al pin 0 e così via. Il valore di ogni bit del registro determina se il pin ad esso corrispondente è utilizzato come ingresso o come uscita. Il valore 1 configura il pin come ingresso ed il valore 0 lo configura come uscita. Ad esempio se si assegna il valore 00000100 al registro TRISB il terzo pin della porta B (RB2) è configurato come ingresso e tutti gli altri come uscite. Un pin configurato come ingresso è collegato a massa tramite un circuito di alta impedenza, mentre uno configurato come uscita è collegato ad un circuito di latch.

Ad ogni porta è associato anche un registro dati indicato con PORT seguito dalla lettera che indica la porta, ad esempio PORTA è il registro dati della porta A. Per ogni pin configurato come uscita il valore del bit del registro corrispondente determina il valore inviato al pin. Viceversa per i pin configurati come ingressi i valori dei bit corrispondenti sono uguali al valore alto o basso rivelato sul pin.

Cinque dei pin della porta A ed i tre pin della porta E sono configurabili anche come *ingressi analogici*. Il registro denominato ADCCON1 permette di definire quali degli otto pin sono utilizzati come ingressi analogici. I registri TRISA e TRISE controllano anche in modo analogico la configurazione di input o output dei pin. Quindi, quando si utilizzano alcuni dei pin come ingressi analogici, è necessario impostare correttamente il valore di questi registri. Come vedremo in seguito, gli ingressi ana-

logici sono collegati ad un convertitore analogico-digitale interno al microcontrollore, che sarà discusso in seguito.

3.5.5 Le memorie

Il microprocessore contiene tre tipi di memoria: la memoria di programma di tipo [8], la memoria di dati volatile di tipo RAM ed una seconda memoria di dati non volatile di tipo EEPROM. Queste memorie utilizzano bus separati denominati Data Bus e Program Bus rispettivamente mostrati in figura 3.38. Le caratteristiche particolari di ogni memoria, descritte qui di seguito, sono determinate dal particolare tipo di funzioni che essa svolge.

- La memoria di programma
  Il "Program Counter", il cui ruolo è descritto in sezione 3.3.6, nel PIC®16F877 ha una profondità pari a 13 bit. Esso è quindi in grado di indirizzare 8000 diversi indirizzi (per la precisione 8191). La memoria di programma del PIC®16F877, costituita da una memoria di tipo , ha quindi una dimensione pari a 8000 parole di 14 bit. 14 bit è anche la larghezza del bus di programma (in figura 3.38 si può vedere la connessione tra la memoria FLASH, il program counter e l'instruction register). La memoria FLASH è una memoria non volatile ed elettricamente cancellabile. L'organizzazione della memoria di programma è mostrata in figura 3.40. Le locazioni di memoria da 0005h ("h" indica la notazione esadecimale) in poi sono divise in quattro banchi di 2000 parole ciascuno. Questa struttura viene sfruttata nella procedura di indirizzamento.

  Le locazioni di memoria 0000h ("reset vector") e 0004h ("interrupt vector") contengono rispettivamente i puntatori alle routine che vengono eseguite al reset del microprocessore ed alla ricezione di un interrupt.

  Il programma che si vuole implementare viene normalmente scritto in Assembler, C o Basic e viene quindi tradotto in linguaggio macchina dai compilatori appropriati. Il programma compilato deve poi essere caricato nella memoria di programma.

- La memoria RAM
  Questa memoria, descritta in dettaglio in sezione 2.2, è utilizzata dalla CPU, dalle periferiche e dall'utente per le variabili di programma. Questo tipo di memoria è caratterizzata da una scrittura molto rapida ($\simeq$ 100 ns) tuttavia le informazioni in essa immagazzinate non vengono conservate quando il microcontrollore non è alimentato, cioè la memoria è volatile. Essa è suddivisa in quattro sezioni dette *banche* ognuna composta da 128 byte (ad esempio gli indirizzi della prima banca vanno da 00h a 7Fh). Le locazioni di memoria più basse di ogni banca sono riservate ai registri per le funzioni speciali che servono a definire la configurazione del microcontrollore o ad istruirlo ad eseguire determinate operazioni. Viceversa le locazioni di memoria più alte sono

[8] Le memorie FLASH sono una sottovarietà delle EEPROM. A differenza di una normale EEPROM, in una *flash memory* non è possibile cancellare e riscrivere singoli bytes, ma solo l'intera memoria o gruppi di bytes

Figura 3.40: Schema della memoria [12].

a completa disposizione dell'utente. In figura 3.41 è mostrato lo schema di organizzazione della memoria RAM in quattro banche e sono indicati i registri dedicati ad uso speciale, ai quali è assegnato un nome mnemonico che aiuti a ricordarne la funzione, ed i registri di uso generale. Si possono ad esempio individuare i registri speciali TRISA o ADCCON1 le cui funzioni sono state precedentemente discusse.

Quando si programma in un linguaggio diverso dal linguaggio macchina è il compilatore che si occupa della gestione della memoria RAM.

- La memoria EEPROM
  Il microprocessore contiene un blocco di 256 byte di memoria di tipo EEPROM. Questa è elettricamente cancellabile, riscrivibile e non volatile quindi verrà utilizzata per tutti i dati che non devono essere persi quando il microcontrollore viene spento. La richiesta di scrittura di un byte in questa memoria procede

Figura 3.41: Schema della memoria RAM. In figura si puó notare la suddivisione in quattro banche di dimensione 128 byte ciascuna (ogni casella indica 1 byte). Le locazioni di memoria piú basse di ogni banca sono dedicate ai registri speciali indicati in figura dalle varie sigle. Le locazioni di memoria piú alte sono dedicate ai registri generici (indicati in figura come General purpose register). I numeri indicati in esadecimale accanto alle locazioni di memoria indicano il relativo indirizzo [12].

attraverso l'operazione di cancellazione e quindi riscrittura del byte. Un ciclo di scrittura viene eseguito in circa 4 ms.

3.5.6 Le periferiche interne

In questa sezione sono brevemente descritte le funzionalità offerte dalle periferiche interne del microprocessore ed il modo per utilizzarle. La descrizione non è esaustiva ma si propone soltanto di dare un'idea delle potenzialità del microprocessore. Le periferiche sono:

- tre contatori indicati come, timer 0 1 e 2;
- un convertitore analogico-digitale (CAN) a 10 bit;
- un modulo per la generazione di impulsi di periodo regolabile (PWM);
- un modulo per la comunicazione seriale sincrona;
- un modulo USART[9];
- un modulo di comunicazione parallela;

Se si analizza lo schema dell'architettura del microprocessore PIC®16F877 mostrato in figura 3.38 si possono facilmente individuare le varie periferiche nella parte inferiore del disegno connesse al bus dei dati. Le caratteristiche principali di queste periferiche vengono descritte qui di seguito.

- I contatori
  I tre contatori (o timer), indicati in figura come TIMER0, TIMER1, TIMER2, possono essere incrementati ogni volta che viene generato un determinato numero di cicli dal segnale di clock del microprocessore. Il numero di cicli necessari per incrementare di una unità i contatori può essere definito attraverso un opportuno registro. Considerando che un ciclo di clock può corrispondere fino ad un minimo di 50 ns, si capisce che i contatori possono essere utilizzati per eseguire misure di tempo piuttosto precise.

  I timer 0 e 2 sono contatori a 8 bit mentre il timer 1 è a 16 bit. Questo significa che i primi due possono contare fino ad un massimo di 255 conteggi mentre l'ultimo fino ad un massimo di 65535 conteggi. I timer 0 ed 1 possono essere incrementati anche attraverso un segnale esterno collegato al pin RA4. Tre registri denominati OPTION_REG, T1CON e T2CON permettono di configurare tutte le opzioni dei tre contatori.

  Ogni volta che uno dei contatori passa dal massimo numero registrabile a 0 può essere generato un segnale di interrupt. Questo segnale permette quindi di contare quante volte il contatore ha ricominciato a contare da 0. Utilizzando, ad esempio, una variabile che viene incrementata ogni volta che l'interrupt è generato, si può estendere l'intervallo di conteggi eseguibili e quindi gli intervalli di tempo misurabili.

- Il convertitore analogico-digitale
  Il convertitore analogico-digitale (ADC) permette di convertire una tensione analogica in un valore numerico binario. La tensione in ingresso all'ADC viene

[9]Universal Synchronous/Asynchronous Receiver/Transmitter

letta da uno degli otto input analogici descritti nella sezione 3.5.4. Tre bit del registro ADCON0 permettono di selezionare quale di questi input è collegato all'ADC, in questo modo è possibile misurare fino ad otto diverse grandezze fisiche con un solo ADC. Lo schema del circuito di ingresso del convertitore analogico digitale è mostrato in figura 3.42.

Figura 3.42: Schema del circuito di ingresso dell'ADC [12].

L'ingresso analogico selezionato è collegato al condensatore di un circuito di sample-and-hold. Questo circuito permette di caricare il condensatore al valore della tensione in ingresso e di mantenere il valore di tensione pressoché costante durante il tempo necessario ad eseguire la conversione. L'uscita del circuito di sample-and-hold è collegata all'ingresso del convertitore che fornisce in uscita il valore digitale attraverso una serie di approssimazioni successive. La tensione è così convertita in un numero a 10 bit.

Per fare in modo che il convertitore funzioni correttamente è necessario che il condensatore di ingresso abbia il tempo necessario per caricarsi al valore di tensione applicata e che in seguito il convertitore abbia il tempo necessario ad effettuare la conversione. Il tempo necessario affinché queste due operazioni siano eseguite è pari a qualche centinaia di microsecondi.

La risoluzione dell'ADC, pari a 10 bit, permette di raggiungere, su un intervallo $0-5\ V$, una precisione di $5\ mV$. I valori di riferimento per le tensioni minima e massima (gli estremi dell'intervallo di conversione) sono programmabili.

- Il modulo CCP
  Il modulo CCP offre tre tipi di funzionalità: Capture, Compare o PWM (Pulse Width Modulation). Quando il modulo funziona in modo "Capture" il valore del timer1 viene letto ogni volta che un determinato evento, definito dall'utente, si realizza. Il tipo di evento può essere il fronte di discesa o di salita

di un segnale o l'occorrenza di un predeterminato numero di fronti di salita. Il tipo di evento è selezionato attraverso il valore di un registro dedicato. Il funzionamento nel modo "Compare" permette di confrontare continuamente il valore dei registri CCPR1H e CCPR1L con il valore del contatore 1 memorizzato nei registri TMR1L e TMR1H (vedi figura 3.41). Ogni volta che i valori sono uguali il pin RC2/CCP1 viene portato ad un valore definito dall'utente. L'utente può scegliere di portare il pin ad un valore alto, basso o di lasciarlo invariato. Anche in questo caso la scelta dell'azione da eseguire viene definita dal valore di un apposito registro. L'ultima modalità di funzionamento è quella come generatore di impulsi (PWM) di periodo e duty-cycle programmabili con una risoluzione fino a 10 bit. Questo modulo è realizzato in modo tale da permettere di generare treni di impulsi in modo molto semplice senza sovraccaricare il microprocessore.

- I moduli per la comunicazione
  Esistono tre moduli dedicati alla comunicazione tra il microprocessore ed il mondo esterno. Due di questi moduli utilizzano la comunicazione seriale, trasmettono cioè i dati su una sola linea, e sono il modulo MSSP (Master Synchronous Serial Port) ed il modulo USART (addressable Universal Synchronous Asynchronous Receiver Transmitter). Queste periferiche permettono di comunicare con altri microcontrollori o con periferiche esterne quali shift registers, driver per schermi, convertitori A/D o D/A. Il modulo USART può essere configurato sia per utilizzare un protocollo di comunicazione "full duplex asincrono" sia "half duplex sincrono". Esiste poi un terzo modulo che permette di utilizzare la porta D come una porta parallela che può essere letta e scritta in modo asincrono. La porta D utilizzata in questo modo può essere direttamente interfacciata ad un bus a 8 bit di un microprocessore esterno.

3.5.7 Gli interrupt

Nella sezione 3.3.4 è stata discussa la procedura generale che porta alla generazione di un interrupt. Il microcontrollore PIC®16F877 è capace di generare interrupts innescati da 14 possibili sorgenti. Ogni volta che si verifica un interrupt il PIC® interrompe l'esecuzione del programma e salta all'istruzione contenuta all'indirizzo 0x04[10] nella memoria di programma. Il tempo che trascorre tra il verificarsi dell'evento collegato all'interrupt e l'istante in cui inizia l'esecuzione dell'istruzione alla locazione 0x04 è detto "latenza di interrupt". Nel PIC®16F877 questa è pari a 3-4 cicli di macchina, a seconda del tipo di interrupt e del punto di esecuzione del programma al momento della ricezione dell'interrupt.

Tra tutte le possibili sorgenti di interrupt quelle che sono utilizzate più frequentemente sono generate da:

- timer 0 e timer 1 quando si ha la transizione dal massimo conteggio raggiungibile, 255 per il timer 0, 65535 per il timer 1, allo zero. Questo interrupt è, ad esempio, utilizzato per realizzare orologi che misurino intervalli di tempo maggiori di quelli ottenibili da un contatore semplice;

[10] A partire da questo indirizzo è memorizzata la Interrupt Service Routine (ISR)

- un fronte di salita o discesa di un segnale collegato al pin RB0. Questo interrupt può ad esempio permettere di rendere sensibile il microprocessore ad un'azione su di un modulo esterno come una tastiera o un pulsante
- la fine della conversione eseguita dal convertitore analogico-digitale;
- la presenza di un segnale sulla porta di comunicazione seriale o su quella parallela.

Ogni sorgente di interrupt può essere singolarmente abilitata o disabilitata.

3.5.8 Programmazione del dispositivo

Come già accennato, il programma sviluppato dall'utente, dopo la compilazione, deve essere trasferito nella memoria programma (FLASH) del PIC®16F877. Allo scopo esistono vari metodi:

- **Programmatore esterno:** In commercio esistono vari modelli di programmatori, sviluppati dalla stessa Microchip® o da terze parti. In generale si tratta di dispositivi piuttosto costosi. Per programmare il microprocessore è necessario rimuovere questo dalla scheda dove è installato, piazzarlo sul programmatore, facendo uso di uno zoccolo, e avviare il programma che gestisce il programmatore stesso. In generale questa tecnica è piuttosto faticosa, in quanto rimuovere il componente è talvolta un'operazione delicata e sicuramente non immediata.
- **Bootloader:** In questo caso la programmazione avviene "in situ", utilizzando la periferica USART. Questa, nota anche come porta seriale, è una interfaccia standard presente in moltissimi elaboratori (compreso il PC), ne risulta che , in questo caso, per programmare il microprocessore non è necessario possedere alcun dispositivo aggiuntivo. In effetti, andando nei dettagli, un programmatore esterno è ancora necessario, ma solo per programmare il PIC® la prima volta, ossia per trasferire nella FLASH un piccolo programma che prende il nome di bootloader. In particolare:
  - Il bootloader viene trasferito dal programmatore nella parte finale della memoria . Necessita soltanto di 256 words, quindi viene scritto nelle locazioni 0x1F00-0x1FFF. Viene anche scritta la locazione 0x0, ossia il reset vector, in modo da puntare alla locazione 0x1F00.
  - All'accensione, quindi, il controllo del PIC® passa subito al programma del bootloader, che attende per 512 ms l'arrivo di un particolare messaggio dalla porta seriale.
  - Se tale messagggio arriva, il bootloader capisce che l'utente vuole trasferire un nuovo programma nella memoria FLASH. D'ora in poi ogni word ricevuta viene scritta nella FLASH, a partire dalla locazione 0x05. Questo equivale quindi a programmare il PIC®.
  - Se il messagggio atteso dal bootloader non arriva entro 512 ms dall'accensione, il bootloader esegue un salto all'istruzione 0x5, dando quindi inizio all'esecuzione del programma dall'utente.

In definitiva, quindi, l'utilizzo del bootloader semplifica molto la fase di programmazione, ma presenta anche due effetti collaterali: il primo è che l'utente non può piu' utilizzare le ultime 256 locazioni (circa il 3% del totale) mentre il secondo è che all'accensione il programma dell'utente viene tenuto in sospeso per circa mezzo secondo, il che potrebbe anche non essere accettabile a seconda dell'applicazione.

3.6 I codici operativi del PIC®16F877

3.6.1 Opcodes

Il circuito integrato PIC®16F877 [12] appartiene alla famiglia dei microcontrollori RISC (Reduced Instruction Set Computer). Questo significa che il numero di codici operativi supportati è abbastanza limitato (soltanto 35) ma che questi sono tutti altamente ottimizzati in termini di velocità di esecuzione. In particolare tutti i codici operativi vengono eseguiti in un solo ciclo macchina (pari a 4 cicli dell'oscillatore esterno) escluse le istruzioni di salto alle funzioni e di ritorno dalle funzioni al programma principale, che prevedono una modifica del valore del Program Counter, e che necessitano di due cicli macchina. D'altra parte l'assenza di codici operativi molto specializzati fa sì che generare codice per funzioni anche molto semplici può richiedere molte istruzioni, il che allunga ovviamente il tempo di esecuzione del programma. Ad ogni modo sicuramente la programmazione in Assembler dei microprocessori RISC è molto semplificata rispetto a quelli tradizionali (CISC: Complex Instruction Set Computer)

Nel PIC® 16F877 ogni istruzione è definita da una parola di 14 bit (pari alle dimensioni del *Program Bus*). Alcuni bit sono l'opcode vero e proprio, che specifica il tipo di istruzione, mentre gli altri sono operandi che specificano ulteriormente il tipo di funzione o includono valori di costanti necessarie all'esecuzione.

Il *Data Bus*, invece, trasporta i dati, provenienti dalla memoria RAM o dalle periferiche e che sono manipolati dai codici operativi. Il data bus e' largo 8 bit, da cui la classificazione del 16F877 come "processore ad 8 bit".

Le istruzioni sono in genere catalogate in tre gruppi:

- **Riferite al byte**: sono istruzioni eseguite per lo più nella ALU, e che eseguono operazioni logiche quali AND,OR,XOR,SHIFT,etc. Quasi tutte fanno uso del registro W, che e' l'equivalente nei processori PIC® del registro accumulatore, descritto in precedenza.

- **Riferite al bit**: sono istruzioni che permettono di modificare singoli bit dei registri, oppure di saltare l'istruzione che segue a seconda del valore di un determinato bit.

- **Letterali e di controllo**: sono istruzioni che hanno al loro interno definito anche il valore di una costante. Tale valore puo' essere utilizzato, a seconda della istruzione, ad esempio come indirizzo dove eseguire salti ad altre parti del programma oppure per essere sommato al registro accumulatore.

E' importante notare che tutti i codici operativi possono agire sia sui registri della RAM di uso generale, e quindi manipolare i dati dell'utente, sia sulle locazioni riservate alla definizione delle periferiche, definendone quindi il comportamento.

Quasi tutti i codici operativi modificano a seguito della loro esecuzione alcuni bit di un registro particolare chiamato *STATUS REGISTER*, localizzato all'indirizzo 0x3 della RAM.
In particolare i bit interessati sono:

- **Bit 0 (C)** : bit di riporto o carry. Si porta al livello alto se l'operazione ha comportato un riporto. Ad esempio una istruzione di somma quale 0x8C + 0x84 = 0x110, poiché il risultato è maggiore di 0xff (che è l'intero ad 8 bit piu' grande) da come risultato 0x110 - 0xff = 0x11, ed inoltre porta a livello alto il bit di carry.
- **Bit 1 (DC)** : molto simile al bit di carry solo che si riferisce al riporto del quarto bit meno significativo (0xf + 0x1 = 0x10 ed inoltre porta alto il bit DC
- **Bit 2 (Z)**: bit di zero, diviene alto se il risultato di una operazione matematica e' 0

La figura 3.43, tratta dal datasheet del microprocessore, riporta schematicamente l'elenco dei codici operativi, sia in forma mnemonica sia come codice binario vero e proprio. Andando nei dettagli:

- **W**: e' il registro accumulatore
- **f**: rappresenta l'indirizzo di una locazione di memoria da utilizzare durante l'esezuzione del'istruzione
- **d**: e' uno dei 14 bit dell'istruzione ed indica dove deve essere scritto il risultato dell'operazione. Se d=0 il risultato viene scritto nel registo W, altrimenti nella locazione all'indirizzo f.
- **b**: indica la posizione di un bit all'interno di un registro
- **k**: indica una costante larga 8 bit da utilizzare nell'istruzione

3.6.2 Esempi di codici operativi

Illustriamo adesso i dettagli relativi a qualcuno dei codici operativi. Le informazioni relative agli altri possono essere facilmente dedotte dalla figura 3.43 oppure consultando il datasheet.

- **Riferite al byte**
  - **ADDWF**: esegue l'addizione tra il contenuto del registro W e il contenuto della locazione f. Il risultato viene scritto nuovamente nel registro W se d=0 altrimenti nel registro f. Le istruzioni sono molto ottimizzate: in un solo ciclo macchina e' possibile leggere un registro, eseguire un'operazione e scrivere di nuovo lo stesso registro. Se il risultato della somma e' superiore a 0xff il bit di carry si porta alto.
  - **DECF**: decrementa di 1 il contenuto del registro f. Se il risultato e' 0 si alza il bit Z dello STATUS REGISTER.

Figura 3.43: Tabella codici operativi 16F877 [12].

  - **NOP**: nessuna operazione viene eseguita

- **Riferite al bit**
  - **BCF**: pone a 0 il bit b del registro all'indirizzo f
  - **BTFSC**: verifica il valore del bit b del registro f. Se questo e' 0 viene saltata l'esecuzione dell'istruzione che segue.

- **Letterali e di controllo**
  - **ADDLW**: somma il contenuto della costante k al registro W e scrive il risultato in W. Sia il bit di carry che il bit Z possono essere modificati dall'operazione.
  - **MOVLW**: copia la costante k nel registro accumulatore W
  - **CALL**: salta all'indirizzo contenuto nella costante k, che e' supposto essere l'indirizzo di partenza di una funzione. Prima pero' salva il contenuto del program counter nello stack, in modo tale che al termine della funzione il programma possa riprendere l'esecuzione a partire dal punto in cui era stata interrotta. Per l'esecuzione richiede 2 cicli di macchina.
  - **GOTO**: simile a CALL, esegue un salto incondizionato ma senza copiare il program counter nello stack. Per l'esecuzione richiede 2 cicli di macchina.

– **RETURN**: eseguita al termine di una funzione. Legge dallo stack l'ultima locazione salvata in modo da poter continuare l'esecuzione a partire dal punto dove era rimasta. Per l'esecuzione richiede 2 cicli di macchina.

3.6.3 Cenni sul linguaggio Assembler

Come già descritto nelle precedenti sezioni, l'Assembler è un linguaggio che assegna ad ogni codice operativo un nome "mnemonico" tale da ricordare il linguaggio umano e quindi semplificare la programazione rispetto a scrivere il programma del microprocessore in termini di 0 e 1 binari.

Oltre alle istruzioni "mnemoniche" relative ai codici operativi, esistono poi delle "direttive"al compilatore (come d'altronde in tutti i linguaggi di programmazione) ossia delle istruzioni che permettono, tra le altre cose, di scrivere codice con maggiore leggibilità.

Vediamo un frammento di codice:

```
;--Carica il registro W in testreg and verifica il valore del bit 2 di testreg

testreg equ 0xC
label1 movwf testreg ; Copia W in testreg
       btfss testreg,2 ; testa il bit 2 di testreg, salta se settato
```

La prima riga , cominciando per ";" , definisce un commento. La seconda assegna il valore *0xc* alla parola *"testreg"* , per aumentare la leggibilità. La terza contiene il codice operativo *movwf*, che copia il contenuto del registro *W* nel registro alla locazione *0xC*, identificata da *testreg*. Questa riga è proceduta dall'etichetta *label1*, che serve per riferirsi a questa riga di programma da parte di altre istruzioni. L'ultima riga contiene l'istruzione *btfss* che verifica se il secondo bit di *testreg* è 1 ed eventualmente salta l'istruzione che segue. Si noti l'indentazione, obbligatoria, presente nel codice riportato.

In genere tutte le dichiarazioni *"equ"* dei registri di configurazione del microprocessore sono contenute in un file di configurazione, fornito insieme al programma assemblatore ed incluso nel codice scritto dall'utente.

3.6.4 Esempio completo di un programma Assembler

Vediamo adesso un esempio di un programma Assembler completo. Si tratta di un generatore di onde quadre. Viene generato un segnale sul Pin 7 della Porta B con una frequenza di circa 5 Hz. Per ogni riga è riportato un commento nel seguito. Il programma è composto dal ciclo principale e dalla funzione *"delay"* che genera il ritardo.

```
1:     ;--Programma "Flash.asm"
2:     ;--Date: 10 agosto 2004
3:     ;--Versione: 1.0
4:        LIST p=PIC16F877, r=hex, f=INHX8M

5:        include "P16F877.inc"
```

```
6:     delreg equ 0x30 ; Etichette per i registri
7:     count equ 0x31  ; dei ritardi

8:        org 0x0

9:    ;--Bit 7 di PortB e' definito come output
10:       bsf STATUS,RP0 ; Seleziona banco 1
11:       bcf STATUS,RP1 ; Seleziona banco 1
12:       bcf TRISB,7    ; resetta il bit 7 del registro TRISB
13:       bcf STATUS, RP0; Seleziona banco 0

14:    ;--Qui comincia il programma principale
15:    begin bsf PORTB,7 ; Alza il pin 7 della porta B
16:       call delay    ; chiama la funzione del ritardo

17:       bcf PORTB,7   ; Resetta il pin 7 della porta B
18:       call delay    ; chiama la funzione del ritardo
19:       goto begin    ; torna all'inzio del programma
20:    ;--Fine del programma principale


21:    ;--Subroutine che si occupa di generare il ritardo

22:    delay movlw 0xFF
23:       movwf delreg ; Carica il registro delreg
24:       movlw 0x82
25:       movwf count ;  Carica il registro count
26:    loop2 decfsz count,1 ; loop esterno
27:       goto loop1
28:       goto exit
29:    loop1 decfsz delreg,1 ; loop interno
30:       goto loop1
31:       goto loop2
32:    exit return
```

Esaminiamo ora, una alla volta, queste istruzioni.

- **1-3**: commenti
- **4**: il processore per cui viene effettuata la compilazione è il 16F877
- **5**: file con le definizioni dei registri
- **6-7**: i registri alle locazioni 0x30 e 0x31 sono riferiti come *delreg* e *count*
- **8**: le istruzioni che seguono sono caricate nelle locazioni di memoria programma (flash) a partire dalla 0x0. La locazione 0x0 corrisponde al vettore di reset, ossia l'esecuzione del programma, all'accensione comincia da qui.
- **10-13**: Il pin 7 della porta B deve essere definito come output. Questo si ottiene ponendo a 0 il bit 7 del registro TRIS B. Questo registro e' localizzato

nel banco 1 della RAM. Prima di tutto quindi, selezioniamo questo banco. Allo scopo i bit RP0 e RP1 del registro di stato devono essere posti, rispettivamente, a 1 e 0. A questo punto possiamo porre a 0 il il bit 7 del registro TRIS B, ed infine selezionare di nuovo il banco 0 di memoria (RP0 = 0)

- **15**: Da questa riga ha inizio il programma principale, detto **main**,
  Viene posto a 1 il bit 7 del registro PORT B, il che equivale a far generare un livello alto al PIN B7 del microprocessore. A questa istruzione viene assegnata l'etichetta *begin*.

- **16**: viene chiamata la funzione *delay*, che genera un ritardo di circa 100 ms

- **17**: viene riportato a 0 il bit 7 del registro PORT B e quindi anche il valore di tensione sul pin B7 si porta a 0

- **18**: viene di nuovo generato un ritardo di 100 ms

- **19**:l'esecuzione salta indietro all'esecuzione del codice operativo con l'etichetta *begin*. E' chiaro quindi che siamo in presenza di un loop infinito

Vediamo adesso in dettaglio come funziona la funzione *delay*. L'idea è quella di eseguire due loop, uno dentro l'altro, decrementando ad ogni ciclo il valore di un registro per loop. Quando entrambi i registri sono 0 l'esecuzione torna al programma principale.

- **22-25**: i registri *delreg* e *count* sono caricati rispettivamente con 0xff e 0x82. Si noti che per caricare un registro con una costante, è necessario prima di tutto caricare la costante nell'accumulatore e poi copiare l'accumulatore nel registro.

- **26-28**: l'istruzione, etichettata come *loop2*, decrementa il valore del registro count, scrivendo il risultato nuovamente dentro count. Se il valore di count è diverso da 0 viene eseguita la riga seguente e quindi il salto al loop interno (*loop1*), se invece è 0 viene saltata una istruzione, quindi si passa alla riga 28, che fa saltare alla riga 32 che esegue l'istruzione di ritorno al ciclo principale. Questo indica che il ritardo programmato è trascorso.

- **29-31**: qui si esegue il loop interno. Il registro *delreg* viene decrementato. Se *delreg* è diverso da 0 si esegue l'istruzione 30, che salta di nuovo indietro alla riga 29. Se *delreg* vale 0 viene saltata una riga, riportando quindi l'esecuzione all' indirizzo *loop 2*, cioè al loop esterno.

Cerchiamo di valutare il ritardo in termini di numero di cicli macchina. Le righe 22-25 sono eseguite in 4 cicli (una per ciclo). Le righe 29-31 costituiscono il loop interno. L'istruzione *decfsz* richiede un ciclo se il risultato è diverso da 0 , altrimenti 2. Le istruzioni *goto* richiedono due cicli, quindi 255 * (1+2) + 2 +2 = 769. Tornando all'istruzione alla riga 26, questa esegue 0x82 = 130 volte il loop secondario. Quindi 130 * (1 +2 +749) + 2 +2 = 97764. Supponendo un clock con frequenza di 4 MHz, e quindi con 1 MHz di frequenza effettiva di esecuzione dei codici operativi, questo comporta un ritardo di circa 100 ms, e poiche' lo stesso

ritardo e' applicato per la parte alta e bassa del ciclo, questo equivale a generare un segnale con frequenza di circa 5 Hz.

Dopo aver compilato il programma con l'assemblatore, si ottiene la seguente lista di codici operativi, espressi in formato esadecimale. Questa stessa lista può poi essere caricata sul microprocessore. Si noti che l'intero programma occupa solamente 20 locazioni della memoria programma.

Utilizzando la tabella in figura 3.43 è possibile riconoscere tutte le istruzioni espresse in formato mnemonico nel listato precedentemente commentato.

```
INDIRIZZO   DATO

0x000000    0x1683
0x000001    0x1303
0x000002    0x1386
0x000003    0x1283
0x000004    0x1786
0x000005    0x2009
0x000006    0x1386
0x000007    0x2009
0x000008    0x2804
0x000009    0x30FF
0x00000A    0x00B0
0x00000B    0x3082
0x00000C    0x00B1
0x00000D    0x0BB1
0x00000E    0x2810
0x00000F    0x2813
0x000010    0x0BB0
0x000011    0x2810
0x000012    0x280D
0x000013    0x0008
```

3.6.5 Esempio di programmazione in C

Risulta interessante valutare lo stesso programma scritto però in un linguaggio di alto livello, quale il C.

```
void main()
 {

   set_tris_b(0x7f);
   while(1){
      output_high(PIN_B7);
      delay_ms(100);
      output_low(PIN_B7);
      delay_ms(100);
           }
 }
```

Ovviamente questo programma risulta molto piu' leggibile e intuitivo del codice Assembler prima analizzato, è peró necessario notare che il codice, dopo la compilazione, richiede 51 locazioni della memoria programma, quindi più del doppio di quanto richiesto dalla programmazione diretta in linguaggio macchina.

Bibliografia

[1] Micro-Cap Evaluation 7.0.6.0. Copyright(c) 1988-2001 Spectrum Software. By Andy Thompson, Tim O'Brien and Bill Steele. Indirizzo del sito Web: http://www.spectrum.co.uk

[2] Donald L. Schilling, Charles Belove: "Electronic circuits, Discrete and Integrated". McGRAW-HILL International Editions (1989). Pagg. 740-745.

[3] Herbert Taub, Donald Schilling: "Elettronica Integrata Digitale". Gruppo Editoriale Jackson (1981). Pagg. 553-562

[4] National Semiconductor Corporation: http://www1.national.com

[5] Paul Horowitz and Winfield Hill: "The art of electronics". Cambridge University Press, 1997

[6] Jacob Millman e Arvin Grabel. "Microelettronica". McGraw- Hill Libri Italia srl, 1987.

[7] Ronald Mancini, Electronic Design, 20 Febbraio 1995, p.110

[8] Donald L. Schilling, Charles Belove: "Electronic circuits, Discrete and Integrated". McGRAW-HILL International Editions (1989). Capitolo 12 John Wiley & Sons 1976 (cap. 8-9)

[9] C.F. Delaney: Electronics for the Physicist with applications, John Wiley & Sons 1980. (cap. 11-13)

[10] Adel S. Sedra and Kenneth C. Smith: "Microelectronic Circuits". 4^{th} Ed., Oxford University Press, 1998

[11] Wolfram Research, Mathematica 4. (www.wolfram.com)

[12] http://ww1.microchip.com/downloads/en/DeviceDoc/30292c.pdf

Indice analitico