4 OTTIMIZZAZIONE DELLE MACCHINE SEQUENZIALI SINCRONE

4 OTTIMIZZAZIONE DELLE MACCHINE SEQUENZIALI SINCRONE 1

4.2 INTRODUZIONE 2
4.3 I BISTABILI: RICHIAMI 6
4.4 IL PROGETTO DEI CONTATORI 12
4.4.1 IL PROGETTO DEI CONTATORI SINCRONI 13
4.4.2 IL PROGETTO DEI CONTATORI ASINCRONI 17
4.5 IL PROGETTO BASATO SU MACCHINE A STATI FINITI 23
4.5.1 LA FSM COME MODELLO DI DESCRIZIONE DI UN CIRCUITO SEQUENZIALE 24
4.5.2 DALLA FSM AL CIRCUITO SEQUENZIALE 33
4.6 LA MINIMIZZAZIONE DEGLI STATI 35
4.6.1 LE MACCHINE COMPLETAMENTE SPECIFICATE 36
4.6.2 LE MACCHINE INCOMPLETAMENTE SPECIFICATE 42
4.7 L'ASSEGNAMENTO DEGLI STATI 49
4.8 MINIMIZZAZIONE LOGICA 52
4.9 LA SINTESI SEQUENZIALE MEDIANTE SIS 54
4.9.1 DALLA FSM AL CIRCUITO SEQUENZIALE 55
4.9.2 LA MINIMIZZAZIONE DEGLI STATI 59
4.9.3 ASSEGNAMENTO DEGLI STATI 63
4.9.4 MINIMIZZAZIONE LOGICA 63

Franco Fummi
Università di Verona

Mariagiovanna Sami
Politecnico di Milano

La presente dispensa è stampabile e riproducibile esclusivamente per gli scopi didattici dei corsi del Politecnico di Milano e dell'Università di Verona; ogni altro uso deve essere preventivamente autorizzato dagli autori.

Febbraio 2000

4.2 INTRODUZIONE

Come si è già visto nei corsi precedenti, una *macchina sequenziale* è caratterizzata dall'essere dotata di *memoria*: il comportamento della macchina in risposta a una configurazione di valori d'ingresso dipende non solo dagli ingressi, ma anche dalla "storia" precedente della macchina - dalla *sequenza* di configurazioni d'ingresso applicate in precedenza. Nella versione più diffusa (e anche più semplice da realizzare) le macchine sequenziali fanno riferimento al *tempo*: si tratta di macchine *sincrone*, che "campionano" le configurazioni d'ingresso in corrispondenza di un *segnale di sincronismo* e il cui comportamento subisce un'evoluzione solo in corrispondenza dello stesso segnale di sincronismo. Tale segnale è costituito da una forma d'onda periodica (un'onda quadra) di frequenza fissa f e periodo T, e viene detto anche in genere *segnale di clock*. Per le macchine sequenziali che hanno pratico interesse (fisica realizzabilità) il comportamento viene definito in risposta a sequenze d'ingresso anche molto lunghe, a pur sempre *finite*; considerando che in un qualsiasi istante la memoria che la macchina conserva degli eventi passati ne costituisce lo *stato*, ne consegue che considereremo *macchine a stati finiti*. Genericamente, quando la macchina si trova in un dato stato *presente* e riceve una data configurazione d'ingresso, evolve verso uno stato *prossimo* e produce una configurazione d'uscita: se (come accade in questo corso) imponiamo che stato presente e configurazione di uscita, se specificati, siano *univoci* - non esiste cioè la possibilità che da un dato stato presente e per una medesima configurazione d'ingresso la macchina possa subire evoluzioni diverse - la macchina è inoltre *deterministica*. Ci limiteremo appunto allo studio di macchine *deterministiche a stati finiti*.

Uno schema generale di tali macchine è riportato in figura 4.1.

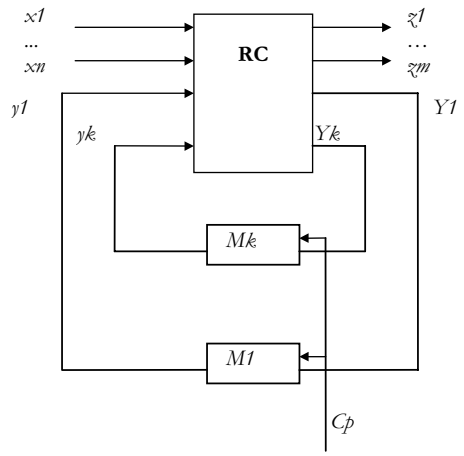


Figura 4.1. *Schema strutturale di una macchina a stati finiti.*

Le variabili *x1…xn* costituiscono gli *ingressi primari* della macchina, di cui le variabili *z1…zm* sono le *uscite primarie*. *M1…k* sono elementi di *memoria* capaci ognuno di mantenere la registrazione di un bit per un istante di clock - in altre parole, sono dei bistabili (nel caso più semplice, sono bistabili di tipo DT), che ricevono tutti lo stesso segnale di clock (*Cp*, il clock principale). L'insieme dei bistabili registra lo *stato* della machina: le variabili *Y1…Yk* codificano lo *stato prossimo* della macchina, mentre le variabili *y1…yk* - che ne codificano lo *stato presente* - sono dette anche *variabili di stato*.

Una macchina deterministica a stati finiti (o più brevemente macchina a stati finiti, spesso indicata con la sigla FSM, *F*inite *S*tate *M*acchine) può anche essere descritta in termini di *teoria degli automi*: possiamo rappresentarla, appunto, come un *automa deterministico a stati finiti*, definito mediante una quintupla M: <*S, I, O, δ, λ*>, dove:

- *S* è l'insieme (*finito e non vuoto*) degli stati. Si noti il particolare "non vuoto": una macchina combinatoria è una particolare macchina sequenziale *con un solo stato* - non esistono macchine prive di stati
- *I* è l'*alfabeto d'ingresso*, costituito dall'insieme di tutte le configurazioni di valori *ammissibili* delle variabili primarie di ingresso. L'alfabeto d'ingresso è finito: può eventualmente essere *vuoto*, nel caso di *macchine autonome*, che evolvono semplicemente in risposta al segnale di clock senza ricevere altra informazione di ingresso.
- *O* è l'*alfabeto di uscita*, costituito da tutte le configurazioni di valori ammissibili delle variabili primarie di uscita. Anche questo alfabeto è finito; in linea di principio, si può anche ammettere che possa essere vuoto - ma una macchina che

non comunica nulla sul proprio stato e il proprio funzionamento è chiaramente priva di significato.

- δ è la funzione di stato prossimo, che - sulla base dello stato presente e del *simbolo d'ingresso* attualmente fornito agli ingressi primari - definisce lo stato prossimo: formalmente, $\delta: S\times I\rightarrow S$. La funzione stato prossimo definisce l'evoluzione della macchina nel tempo, in risposta agli eventi di ingresso.
- λ è la *funzione di uscita* che determina il simbolo dell'alfabeto di uscita che la macchina deve, istante per istante, produrre.

Sono state definite due tipologie di automi deterministici a stati finiti: le *macchine di Mealy* e le *macchine di Moore*. La differenza sostanziale fra le due riguarda proprio la funzione di uscita: mentre nelle macchine di Mealy la funzione d'uscita costituisce la risposta fornita dalla macchina *quando, trovandosi in uno stato presente, riceve un dato simbolo di ingresso* - ed è quindi formalizzabile come $\lambda: S\times I\rightarrow O$, nelle macchine di Moore la funzione d'uscita è la risposta *associata allo stato in cui la macchina si trova*: $\lambda: S\rightarrow O$. Se vogliamo vedere in termini intuitivi la differenza fra i due casi, nella macchina di Mealy "leggiamo" il simbolo di uscita *mentre la macchina subisce una transizione da stato presente a stato prossimo* - in corrispondenza del segnale di clock; nella macchina di Moore, leggiamo il simbolo di uscita *mentre la macchina permane nello stato* - cioè nell'intervallo fra due impulsi di clock successivi.

Per garantire il corretto funzionamento fisico di una macchina sequenziale, è in genere opportuno definire a priori uno *stato iniziale* in cui la macchina si deve portare - ad esempio, quando viene applicata l'alimentazione, o quando viene applicato il segnale di reset. La definizione dell'automa si modifica da quintupla a sestupla, diventando:

M: <*S, I, O, δ, λ, s*>, dove *s* è appunto lo stato iniziale.

Una macchina a stati finiti può essere descritta compiutamente mediante una *tabella degli stati* che indichi, per ogni coppia stato presente-simbolo d'ingresso, lo stato prossimo e il simbolo di uscita (che eventualmente possono essere non specificati). La stessa informazione può essere fornita in forma grafica mediante un *diagramma degli stati*, o *grafo delle transizioni*, in cui ogni nodo è associato a uno stato, un arco *orientato* da uno stato *si* ad uno *sj* e contrassegnato con un simbolo d'ingresso α rappresenta la *transizione* che si verifica quando alla macchina, che si trova nello stato presente *si*, si applica il simbolo d'ingresso α che la porta nello stato prossimo *sj*. Nel caso di macchina di Mealy, all'arco che rappresenta la transizione verrà associato anche un simbolo di uscita; nel caso di una macchina di Moore, per ogni stato il simbolo di uscita sarà *associato al nodo che rappresenta lo stato stesso*. In figura 4.2 sono riportati la tabella degli stati e il grafo delle transizioni per una semplice macchina di Mealy, in figura 4.10 quelli per una semplice macchina di Moore.

	0	1
A	B / 0	C / 1
B	A / 1	C / 0
C	B / 0	A /0

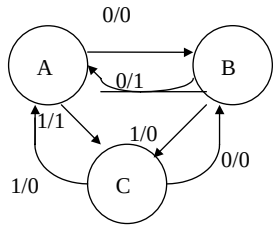


Figura 4.2. *Macchina a stati finiti di Mealy.*

	0	1	z
A	B	C	0
B	A	C	1
C	B	C	1

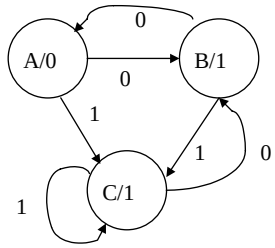


Figura 4.3. *Macchina a stati finiti di Moore.*

Nella fase di *sintesi*, il progettista deve passare dalla specifica (a parole, o in forma algoritmica) della FSM al diagramma degli stati e alla relativa tabella degli stati; questa dovrà quindi essere sottoposta ad una *minimizzazione del numero di stati*, che costituisce una prima fase nel processo di *ottimizzazione* della macchina. Realizzata una tabella che garantisca il comportamento voluto col numero minimo di stati, si

deve passare all'*assegnamento*, cioè alla *codifica* degli stati col minimo numero possibile di variabili interne, e di qui alla sintesi della rete combinatoria RC indicata in figura 4.1. Anche in questi passi sono possibili interventi di ottimizzazione - volti sia a ridurre l'area (semplificando la rete combinatoria RC) sia a ridurre i ritardi.

Il percorso di progetto ora indicato è del tutto generale e più avanti lo si esaminerà in dettaglio, considerando in particolare le tecniche di ottimizzazione e facendo riferimento a questo scopo a un particolare strumento di CAD (il programma SIS, sviluppato all'Università di California a Berkeley). Prima però si ricorderanno brevemente le caratteristiche delle principali classi di bistabili, e si considereranno tecniche di progetto specifiche adatte a particolari classi di dispositivi sequenziali, che permettono di giungere rapidamente a un progetto ottimale senza ricorrere all'intera filiera di progetto ora indicata.

4.3 I BISTABILI: RICHIAMI

Un bistabile è - come dice il nome - un dispositivo capace di mantenersi stabilmente in uno fra due *stati*. La necessità di codificare due soli stati fa si che al bistabile venga associata una sola *variabile di stato*; la codifica dello stato presente coincide col valore dell'uscita del bistabile stesso (o, in altri termini, si legge direttamente dall'esterno il valore dello stato).

Una prima classificazione distingue fra bistabili *asincroni* e bistabili *sincroni*. Nei primi, è la variazione di un segnale presente a uno degli ingressi dati che può determinare l'evoluzione del bistabile imponendone il cambiamento di stato (i bistabili asincroni hanno solo ingressi dati). Nei secondi, le variazioni degli ingressi dati vengono campionate dal segnale presente sull'ingresso di sincronismo, e solo quando tale segnale assume un particolare valore il bistabile può evolvere.

Il più semplice bistabile asincrono è detto *SC* (*S*et-*C*lear, a volte anche *SR*, *S*et-*R*eset): ha due ingressi, *S*e *C*, e una variabile di stato *Q*, e il suo funzionamento può essere riassunto come segue:

- se è presente un 1 sull'ingresso *S* e uno 0 sull'ingresso *C*, lo stato prossimo vale 1, indipendentemente dal valore dello stato presente;
- se è presente uno 0 sull'ingresso *S* e un 1 sull'ingresso *C*, lo stato prossimo vale 0, indipendentemente dal valore dello stato presente;
- se su ambedue gli ingressi è presente valore 0, lo stato prossimo è identico allo stato presente.

Si può notare che non è prevista la configurazione d'ingreso *SC=11*. In effetti, facendo riferimento allo schema di bistabile *SC* in figura 4.4, si verifica che per tale

configurazione degli ingressi il comportamento del bistabile verrebbe a dipendere dalle relazioni fra i ritardi distribuiti lungo i vari percorsi. Chiaramente, un tale comportamento non è accettabile nel caso di produzione in serie; si impone quindi che la configurazione d'ingresso *11* non possa verificarsi.

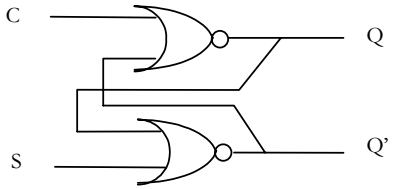


Figura 4.4. *Schema logico di bistabile SC.*

La tabella delle *transizioni* del bistabile è riportata in tabella 4.1a, dove si indica lo stato prossimo *Q** in funzione dello stato presente *Q* e degli ingressi primari *S* e *C*.

	Q*			
Q	**SC=00**	**SC=01**	**SC=11**	**SC=10**
0	0	0	X	1
1	1	0	X	1

Q	**Q***	**S**	**C**
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Tabella 4.1. *(a) Tabella delle transizioni e (b) tabella delle eccitazioni del bistabile SC.*

Oltre alla tabella delle transizioni, per i bistabili è utile conoscere la tabella delle *eccitazioni* che indica - per ogni possibile transizione stato presente - stato prossimo - quali valori di eccitazione si devono applicare agli ingressi dati. La tabella delle eccitazioni si ricava facilmente da quella delle transizioni; ad esempio se lo stato presente è 0 e si vuole che quello prossimo sia ancora 0, dalla seconda e dalla terza colonna di tabella 4.1a si ricava che *S* deve mantenere il valore 0 mentre *C* può assumere sia il valore 0 sia quello 1 (quindi ha valore *indifferente*), se lo stato presente vale 0 e quello prossimo deve valere 1 si deduce (quinta colonna della tabella delle transizioni) che deve essere *S=1* e *C=0*, e così via (tab. tabella 4.1b).

Un altro bistabile asincrono molto usato è quello detto "*T*" (la T è variamente interpretata come l'iniziale di *Trigger* o di *Toggle*) - l'unico con un solo ingresso. Il funzionamento viene descritto molto semplicemente come segue:

- quando si presenta un valore 1 (o, più in generale, *attivo*) sull'ingresso *T*, lo stato del bistabile *commuta* (lo stato prossimo è cioè complementare rispetto a quello presente)

- altrimenti, lo stato rimane immutato.

La tabella delle transizioni del bistabile *T* è riportata in tabella 4.2a, quella delle eccitazioni in tabella 4.2b.

	Q*	
Q	**T=0**	**T=1**
0	0	0
1	1	0

Q	**Q***	**T**
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 4.2. *(a) Tabella delle transizioni e (b) tabella delle eccitazioni del bistabile T.*

Ambedue i bistabili ora descritti sono del tipo detto *trasparente* - un opportuno valore presente sugli ingressi dati si riflette immediatamente nel valore dello stato (e quindi dell'uscita).

Passando ora ai bistabili *sincroni*, il primo di essi si ricava proprio dal bistabuile *SC* aggiungendo un ingresso di sincronismo *T*: il funzionamento cambia come segue:

- quando sull'ingresso *T* è preesente un valore attivo, il bistabile reagisce ai valori presenti sugli ingressi *S* e *C* secondo le regole dette prima;
- se il valore all'ingresso *T* è quello non attivo, il bistabile mantiene immutato il proprio stato per qualsiasi configurazione applicata agli ingressi dati.

Tabella degli stati e tabella delle eccitazioni prima introdotte possono ancora esere considerate valide, a patto di specificare che fanno riferimento alla presenza di un valore attivo sull'ingresso *T*. (Il segnale di sincronismo viene normalmente dedotto dal clock principale, e indicato come C_p). Il bistabile *SCT* può essere ricavato molto semplicemente da quello SC mediante lo schema in figura 4.5.

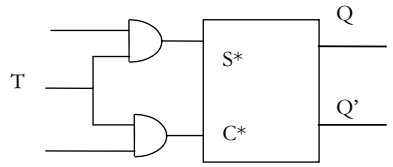


Figura 4.5. *Schema di bistabile SCT.*

Un bistabile sincrono con comportamento simile a quello dello SCT, ma che supera il vincolo riguardante la presenza di due 1 contemporanei sugli ingressi dati, è il bistabile *JKT*, il cui comportamento - *quando su T è presente il valore attivo* -è descritto come segue:

- se è presente un 1 sull'ingresso *J* e uno 0 sull'ingresso *K*, lo stato prossimo vale 1, indipendentemente dal valore dello stato presente;
- se è presente uno 0 sull'ingresso *J* e un 1 sull'ingresso *K*, lo stato prossimo vale 0, indipendentemente dal valore dello stato presente;
- se su ambedue gli ingressi è presente valore 0, lo stato prossimo è identico allo stato presente;
- se su ambedue gli ingressi è presente valore 1, lo stato del bistabile commuta.

Tabella delle transizioni e tabella delle eccitazioni del bistabile *JKT* - limitatamente a valore attivo presente su T - sono riportate in tabella 4.3a e b, rispettivamente.

	Q*			
Q	**JK=00**	**JK=01**	**JK=11**	**JK=10**
0	0	0	1	1
1	1	0	0	1

Q	**Q***	**J**	**K**
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Tabella 4.3. *(a) Tabella delle transizioni e (b) tabella delle eccitazioni del bistabile JK.*

Un ultimo bistabile sincrono molto largamente usato nella sintesi di reti sequenziali è il bistabile *DT* (*D* sta per *data*), descritto molto semplicemente come segue:

- in presenza di valore attivo sull'ingresso *T*, all'uscita del bistabile si trasferisce il valore presente sull'ingresso dati *D*.

La tabella delle transizioni e tabella delle eccitazioni sono molto semplici e sono riportate in in tabella 4.4a e b, rispettivamente.

	Q*	
Q	**D=0**	**D=1**
0	0	1
1	0	1

Q	**Q***	**D**
0	0	0
0	1	1
1	0	0
1	1	1

Tabella 4.4. *(a) Tabella delle transizioni e (b) tabella delle eccitazioni del bistabile DT.*

In particolare, si noti che il valore dell'eccitazione *D* coincide col valore dello stato prossimo voluto (questo, come si vedrà in seguito, semplifica la sintesi della rete sequenziale).

Spesso, per garantire il buon funzionamento di una rete sequenziale i bistabili *trasparenti* non hanno comportamento soddisfacente; si consideri ad esempio la struttura in figura 4.6, costituita dalla catena di tre bistabili *DT* che dovrebbero formare un *registro a scorrimento* (*shift register*), cioè una struttura con un unico ingresso seriale (I_s) e un'uscita seriale (O_s) il cui comportameno è descritto come segue:

- in presenza di un segnale attivo sull'ingresso T, il contenuto dei bistabili *scorre* da sinistra verso destra; l'ingresso seriale I_s viene caricato nel bistabile più a sinistra, mentre il valore registrato nel bistabile più a destra si perde.

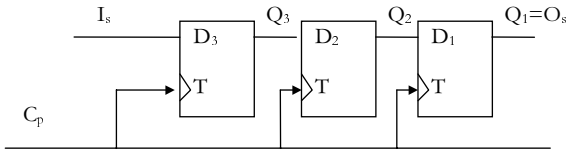


Figura 4.6. *Schema di registro a scorrimento basato su bistabili DT.*

Si supponga che il valore inizialmente presente nei tre bistabili sia (da sinistra verso destra) 101, e si presenti sull'ingresso seriale il valore 0; il coretto funzionamento vorrebbe che, all'arrivo del valore attivo su *T*, la configurazione nel registro diventasse 010. All'arrivo del segnale di sincronismo (qui indicato anche come *segnale di scorrimento*) nel bistabile 3 si carica effettivamente il valore 0; si suppomga però che il valore di *T* resti attivo per un tempo sufficientemente lungo rispetto ai ritardi interni dei bistabili. Il bistabile 2, che aveva inizialmente campionato l'ingresso $D_2=Q_3=1$, si trova a campionare il *nuovo* valore di Q_3, cioè 0, e il valore di Q_2 si porta di conseguenza a 0; lo sttesso può accadere col bistabile 1. *In conclusione - e in dipendenza dai valori relativi dei ritardi intterni dei bistabile e del segnale di sincronismo - il comportamento non è quello corretto e può essere sostanzialmente imprevedibile.*

Per ovviare a questo inconveniente sono stati introdotti i bistabili *con memoria ausiliaria*, detti anche *master-slave* o - più in generale (indipendentemente cioè da un particolare schema logico) *edge sensitive*, che cioè rispondono a un *fronte*, una *transizione di valore* su un particolare ingresso (di fatto, sull'ingresso di sincronismo per i bistabili sincroni). Consideriamo qui la struttura di un bistabile *master-slave* di tipo *DT* che *commuti sul fronte di discesa del segnale* - cioè in corrispondenza della transizione di *T* da 1 a 0. La struttura del dispositivo è quella mostrata in figura 4.7; il bistabile è costituito da due stadi, il *master* e lo *slave*.

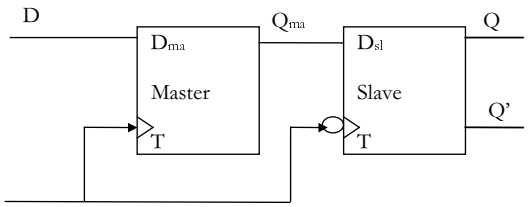


Figura 4.7. *Schema di bistabile master-slave.*

Il segnale di sincronismo viene inviato in forma naturale allo stadio master, in forma negata allo slave. Fino a quando su *T* è presente il valore 0, lo stadio master è disattivato, e all'ingresso *D* dello stadio slave è applicato lo stato del master; lo stato "esterno" del bistabile è quindi identico a quello del master. Quando *T* passa a 1, al master si applica il valore dell'ingresso *D*, che viene trasferito su Q_{ma}; lo slave però è isolato, dato che *T'* vale 0, quindi lo stato globale *Q* non viene modificato rispetto al valore precedente. Quando *T* passa nuovamente a 0 (sul *fronte di discesa* di *T*) lo stato presente del Master viene applicato all'ingresso dati dello slave e quindi trasferito all'uscita globale, mentre il master è nuovamente isolato. L'ingresso esterno *D* si trasferisce quindi all'uscita globale del bistabile sul fronte di discesa del segnale di sincronismo.

Si supponga ora che il registro a scorrimento in figura 4.6 venga realizzato non più con bistabili di tipo trasparente ma con bistabili master slave. Quando sul segnale di sincronismo è presente il valore 1, gli stadi master di tutti i bistabili registrano il valore presente sulle uscite degli stadi slave, mentre questi ultimi stadi sono isolati e quindi rimangono stabili al valore registrati. Quando *T* torna a 0, i master risultano a loro volta isolati e non modificano i valori precedentemente registrati, mentre tali valori vengono trasferiti alle uscite degli stadi slave. Lo scorrimento viene quindi correttamente operato, indipendentemente dai ritardi interni di propagazione dei vari bistabili.

Infine, occorre notare che in molti è necessario poter *forzare* il bistabile in uno stato predeterminato *indipendente dai valori degli ingresi dati o dalle sequenze su tali ingressi*: ciò succede, tipicamente, per garantire che al momento in cui si dà alimentazione al circuito i bistabili in esso presenti si portino in uno stato "iniziale" noto (in genere, 0).Per questo, i bistabili possono essere dotati di un *ingresso asincrono di reset*: portando un dato valore su tale ingresso si forza lo stato iniziale. Da questo momento in avanti, il valore dell'ingresso di reset (che *non* è un ingresso dati) viene mantenuto costante e al valore "non attivo", e quindi l'ingresso non ha più alcuna influenza sul funzionamento del bistabile.

4.4 IL PROGETTO DEI CONTATORI

Il contatore, nella sua definizione più elementare, è un dispositivo sequenziale il cui *stato* registra il numero di impulsi contati in ingresso: riceve in ingresso il segnale di conteggio - l'impulso di clock *Cp* - e in corrispondenza incrementa (se si tratta di un contatore verso l'alto) o decrementa (per un contatore verso il basso) il numero registrato. Le caratteristiche in base a cui si possono distinguere i contatori sono le seguenti.

- Il *modo* di funzionamento - contatore *sincrono* (tutti i bistabili che costituiscono la memoria del contatore ricevono simultaneamente in ingresso il segnale di conteggio e tutti quelli che devono commutare commutano simultaneamente), oppure *asincrono* (almeno uno dei bistabili *non* riceve in ingresso il segnale di conteggio e il suo eventuale cambiamento di stato è determinato solamente dalla sua *funzione stato prossimo* che dipende dallo stato degli altri bistabili).
- Il *codice di conteggio* - cioè la sequenza di configurazioni delle variabili di stato attraverso cui il contatore deve evolvere al succedersi degli impulsi di conteggio; il codice binario naturale è certamente il più diffuso, ma diffusi sono anche i codici decimali (in particolare, il BCD) e, per applicazioni legate a particolari trasduttori di segnali non elettrici provenienti dal mondo esterno, il codice Gray.
- Il *modulo M* secondo cui il contatore deve operare: il contatore in modulo *M* "conta" da 0 a *M-1*, e torna a 0 col successivo impulso di conteggio. Il modulo è essenziale per determinare il *minimo numero di bistabili* necessario per realizzare il contatore: tale numero è infatti pari a $\lceil \log_2 M \rceil$.

Le tre caratteristiche sono "ortogonali" l'una all'altra - possono cioè essere definite in modo reciprocamente indipendente (salvo eccezioni, come si vedrà). Un ulteriore fattore da prendere in considerazione è il *tipo di bistabile* che si vuole usare nel corso del progetto; per un contatore *sincrono* dovremo usare un bistabile sicrono oppure avviare all'ingresso (agli ingressi) dei bistabili la *funzione di eccitazione* che determina il prossimo stato in AND col segnale di conteggio, per un contatore asincrono dovremo certamente usare bistabili di tipo asincrono. Comunque, per garantire il corretto funzionamento, occorre usare bistabili *del tipo master-slave*: infatti, durante lo stesso impulso di conteggio per un generico bistabile dobbiamo usare lo stato presente per determinare le funzioni di eccitazione e registrare nel medesimo bistabile lo stato prossimo.[1]

[1] Nelle librerie di progetto VLSI esistono anche strutture di bistabili accuratamente progettati per garntire questo tipo di funzionamento senza la struttura "a memorua ausiliaria" caratteristica del master-slave: si tratta però di dispositivi accuratamente calibrati a livello di progetto elettrico e di layout, mentre qui si vuole

Consideriamo ora il progetto dei contatori più semplici (e più diffusi), cioè quelli di tipo *sincrono*.

4.4.1 Il progetto dei contatori sincroni

Consideriamo dapprima il caso di un contatore *n codice binario naturale*: il "minimo" possibile è un contatore modulo 2, che evolve lungo la sequenza 0-1-0… Si osserva immediatamente che il bistabile *T*, cui si applichi in ingresso il segnale di conteggio, costituisce esattamente un contatore modulo 2; passiamo ora a un generico modulo 2^n. Sia $n=3$ il ciclo di conteggio, con la relativa sequenza di stati prossimi, è riportato in tabella 4.5 (per semplicità, lo stato prossimo di un bistabile *Q* viene indicato come *Q**).

A	B	C	A*	B*	C*
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Tabella 4.5. *Ciclo di conteggio di un contatore a codice binario naturale.*

Si supponga di usare bistabili di tipo *T* master-slave che commutano *sul fronte di discesa*: le funzioni di eccitazione si costruiscono imponendo che ci sia un fronte di discesa all'ingresso di un bistabile ogni volta che il bistabile deve commutare. Poiché il contatore è *sincrono*, la funzione di eccitazione sarà in AND con il segnale di conteggio *Cp*: quindi per generare il fronte di discesa essa deve valere 1. Si genera quindi la tabella 4.6.

presentare un approccio assolutamente generale.

A	B	C	A*	B*	C*	Ta	Tb	Tc
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

Tabella 4.6. *Tabella delle eccitazioni di un contatore a codice binario naturale.*

La sintesi delle tre funzioni di eccitazione dà:

$Tc=1$

$Tb=C$

$Ta=BC$

L'effettiva funzione applicata all'ingresso del bistabile *i*-esimo è $ti=Ti \bullet Cp$. Si può generalizzare la sintesi vista sopra considerando che - con l'esclusione del bistabile meno significativo - ogni bistabile *i*-esimo commuta tutte le volte che *tutti i bitsabili nelle posizioni da i-1- a 0 hanno stato 1*: questo permette di stabilire immediatamente $T_i=Q_{i-1}….Q_0$, indicando con 0 la posizione del bistabile meno significativo. Abbiamo quindi la possibilità di realizzare un contatore con codice binario naturale di un qualsiasi modulo 2^n; il sistema è effettivamente *sincrono* - tutte le eventuali commutazioni si verificano simultaneamente col fronte (in questo caso, di discesa) del segnale di conteggio.

La regolarità di generazione delle funzioni di eccitazione è legata allo schema di conteggio, non alla tipologia del bistabile; si voglia infatti realizzare un contatore modulo 16 con bistabili DT. La tabella delle accitazioni è in tabella 4.7 (si è tralasciato di elencare esplicitamente gli stati prossimi, dato che per ogni configurazione di stato presente lo stato prossimo non può che essere dato dalla configurazione binaria successiva, tenendo presente che la configurazione *M-1* è seguita dalla configurazione 0).

A	*B*	*C*	*D*	*Da*	*Db*	*Dc*	*Dd*
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

Tabella 4.7. *Tabella delle eccitazioni di un contatore modulo 16.*

La sintesi porta a definire:

$D_d=D'$

$D_c=C'D+CD'=C\oplus D$

$D_b=B'CD+BC'+BD'=B'CD+B(C'+D')=B\oplus(CD)$

$D_a=AB'+AC'+AD'+A'BCD=A(B'+C'+D')+A'BCD=A\oplus(BCD)$

ossia in generale:

$D_i=Q_i\oplus(Q_{i-1}\ldots Q_0)$

Si noti che quanto sopra implica grande facilità di progetto anche in termini di CAD; dato un elemento di libreria, contatori più complessi possono infatti essere realizzati estendendo l'elemento-base. Sia usando bistabili di tipo *T*, sia usando quelli *DT*, in pratica si considera il blocco di bistabile da *i-1* a 0 e si verifica la loro condizione di "traboccamento" che fa passare dal massimo valore conteggiabile allo 0; questa è la

condizione che va in ingresso al bistabile aggiunto in posizione *i*. Sostanzialmente, si crea un segnale di *carry* che viene usato - in modo regolare - per generare l'ingresso al bistabile più significativo.

Si consideri ora il caso di un contatore per una codifica diversa da quella binaria naturale: ad esempio, consideriamo un contatore decimale. Conviene considerarlo scomposto in *blocchi*, corrispondenti ognuno a una cifra decimale: lo schema generale sarà costituito da tanti blocchi quante sono le cifre decimali in *M-1*. Per ogni cifra decimale il blocco contatore richiede un minimo di quattro bistabili (esattamente quattro se, ad esempio, il contatore decimale è per codice BCD o "8421"). Questa soluzione comporta un certo "spreco" in termini di bistabili - un contatore decimale modulo 1000, che conta cioè da 0 a 999, richiede 4×3=12 bistabili, mentre il minimo indispensabile per contare modulo 1000 sarebbe pari a 10 - ma chiaramente offre grande semplicità di progetto: si tratta di progettare il contatore decadico per una cifra decimale, che generi un riporto *c* quando passa da 9 a 0; il segnale di riporto dovrà andare in AND col segnale di conteggio all'ingresso del contatore decadico per la cifra immediatamente più significativa. Si vedrà quindi semplicemente il contatore per una cifra decimale in codice BCD: la tabella delle eccitazioni è riportata in tabella 4.8 per l'ipotesi di bistabili di tipo *DT*.

Q8	**Q4**	**Q2**	**Q1**	**D8**	**D4**	**D2**	**D1**
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

Tabella 4.8. *Tabella delle eccitazioni di un contatore BCD.*

Si noti che questa volta siamo in presenza di una macchina sequenziale in cui il numero degli stati non è una potenza di 2; esistono cioè alcune configurazioni delle variabili di stato (nel nostro caso, 6) che - a meno di malfunzionamento, non si presenteranno mai. Questo permette evidentemente di semplificare la rete

combinatoria che genera le funzioni di eccitazione. Sfruttando al meglio le condizioni di indifferenza così generatesi, le funzioni di eccitazione dei bistabili diventano:

$D_1=Q_1'$

$D_2=Q_8'Q_2'Q_1+Q_8'Q_2Q_1'$

$D_4=Q_8'Q_4'Q_2Q_1+Q_4Q1'+Q_4Q_2'$

$D_8=Q_8Q_2'+Q_4Q_2Q_1$

mentre la condizione di riporto coincide col passaggio del bistabile di "peso" 8 nel codice da 1 a 0, quindi con la condizione 1001 ($Q_8Q_4'Q_2'Q_1$).

Infine, occorre notare che un contatore può essere - come negli esempi indicati - "in avanti" o "verso l'alto", oppure "all'indietro" ("verso il basso") o anche *bidirezionale*. In quest'ultimo caso, il contatore *ha un ingresso primario* aggiuntivo che segnala la direzione del conteggio.

Riprendiamo il caso del contatore binario con modulo 3 realizzato con bistabili T. Per realizzare il contatore "all'indietro" (da 111 a 000) le funzioni di eccitazione sono:

$Ta=1$ $\quad t_a=Cp$

$Tb=C'$ $\quad t_b=C'Cp$

$Ta=B'C'$ $\quad t_c=B'C'Cp$

Per il contatore *bidirezionale*, se il segnale U vale 1 per il conteggio verso l'alto, e 0 per il conteggio verso il basso, le funzioni saranno:

$Ta=1$

$Tb=U\,C+U'\,C'$

$Tc=U\,B\,C+U'\,B'\,C'$

4.4.2 Il progetto dei contatori asincroni

Come si è indicato in precedenza, un contatore *asincrono* è caratterizzato dal fatto che *in almeno uno dei bistabili* che lo compongo non entra direttamente l'impulso di conteggio, ma la commutazione viene generata dal fronte (di discesa o di salita, a seconda delle caratteristiche dei bistabili) che si presenta nella funzione di eccitazione (che, per tali bistabili, *non va in AND con il segnale Cp*). Anche per questi contatori, può essere arbitrariamente specificato *il ciclo di conteggio* (quindi il codice) come anche il *modulo*: può però accadere che per un dato *ciclo di conteggio* il contatore *non possa essere realizzato in quanto asincrono*. Più precisamente, si può verificare il caso che, per garantire tutti i fronti di discesa necessari alla commutazione dei bistabili, le funzioni di eccitazione di *tutti* i bistabili debbano ancora una volta essere poste in AND con il segnale di conteggio.

Per vedere come si affronta il progetto di un contatore asincrono, partiamo ancora

una volta dal caso più semplice ed usuale, quello di un contatore binario in modulo 2^n; come si è già detto, il bistabile T (che, si noti, è un bistabile *asincrono*, dato che fra i propri ingressi non prevede esplicitamente l'ingresso di clock) è un contatore binario modulo 2. Consideriamo un singolo bistabile T: esso genera alla sua uscita un fronte di discesa ogni volta che lo stato passa da 1 a 0 – nella nostra ottica, ogni volta che "trabocca" dal modulo 2. Se pensiamo al singolo bistabile T come a un *blocco elementare* di conteggio binario, la sua uscita può essere vista simultaneamente anche come un segnale di *carry* che va direttamente in ingresso al blocco elementare successivo (se il primo bistabile della catena commuta a ogni impulso di conteggio, il secondo commuterà in corrispondenza del terzo, quinto, settimo impulso, e così via). Più formalmente, riprendiamo la tabella che descrive il ciclo di conteggio di un contatore binario modulo 8 (tabella 4.5). Nelle colonne delle funzioni di eccitazione inseriamo per il momento *tutti e soli i fronti di discesa necessari per avere la commutazione del corrispondente bistabile là dove il ciclo di conteggio la richiede*: ciò implica l'esame per ogni bistabile di una *coppia di righe* successive: se dalla prima alla seconda c'è commutazione, per la funzione di eccitazione dovremo inserire un 1 nella prima riga e (se vogliamo garantire un fronte di discesa *autonomo*) uno 0 nella seconda riga. Si ricordi che il ciclo di conteggio si "richiude" ossia la prima riga rappresenta il valore successivo all'ultimo nel normale funzionamento.

A	B	C	Ta	Tb	Tc
0	0	0	0	0	1
0	0	1		1	1
0	1	0		0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1		1	1
1	1	0		0	1
1	1	1	1	1	1

Tabella 4.9. *Tabella delle eccitazioni di un contatore asincrono.*

Possiamo subito notare che – nella colonna che dà la funzione di eccitazione del bistabile meno significativo – *non compaiono fronti di discesa autonomi:* a ogni impulso di conteggio si richiede una commutazione, quindi in ogni riga si rende necessario l'inserimento di un 1. Per ottenere il fronte di discesa necessario a far commutare il bistabile *C*, occorrerà mandare in AND la funzione di eccitazione *Tc*

con il segnale di conteggio stesso.

Passiamo alla colonna *Tb*: in questo caso abbiamo tutti i fronti di discesa autonomi (sequenze 1-0) richiesti: il bistabile *B* non richiederà al proprio ingresso il segnale di conteggio. Vediamo immediatamente che è *Tb=C*, come peraltro si era visto prima intuitivamente.

Consideriamo ora il bistabile *A*; intuitivamente, vediamo che i suoi fronti di discesa coincidono con quelli del bistabile *B*, quindi il collegamento dell'uscita *B* all'ingresso *Ta* ci garantisce le commutazioni richieste. Una formalizzazione più generale del procedimento richiede che si "completi" in qualche modo la colonna *Ta*: di fatto, quello che si richiede è che – nelle posizioni finora non specificate – non si verifichino *fronti di discesa non voluti*. Abbiamo cioè una situazione particolare di condizioni di indifferenza "vincolate": se in una riga non specificata si assegna alla funzione di eccitazione valore 1, il valore dovrà *restare* 1 fino alla prossima condizione *specificata* che sarà – ovviamente – ancora un 1. Per evidenziare questo vincolo introduciamo delle condizioni d'indifferenza *numerate* con indici crescenti "a ritroso" dall'1 verso il primo 0 *immediatamente precedente*: il vincolo si esprime chiedendo che, se una condizione d'indifferenza vincolata con indice *i* viene posta a 1, tutte le condizioni con indice da *i-1* a 0 abbiano anch'esse valore 1. Nel nostro esempio, la tabella viene completata come in tabella 4.10 (l'insieme di condizioni d'indifferenza vincolate *x'1x'0* deve essere trattato *indipendentemente* da quello *x1x0*).

A	B	C	Ta	Tb	Tc
0	0	0	0	0	1
0	0	1	x1	1	1
0	1	0	*x0*	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	*x'1*	1	1
1	1	0	*x'0*	0	1
1	1	1	1	1	1

Tabella 4.10. *Condizioni di indifferenza per il contatore asincrono.*

La sintesi ottima di *Tc* dà, com'era prevedibile, *Tc=B*. Il contatore binario in modulo 2^n può essere esteso a valore di *n* comunque alto, semplicemente collegando l'uscita dell'ultimo bistabile del contatore in modulo 2^{n-1} all'ingresso di un nuovo bistabile

T: alla semplicità della struttura (certamente superiore a quella del contatore sincrono) si associa però una maggior lentezza rispetto al contatore asincrono – per ottenere la commutazione del bistabile di posizione *n*, e quindi la stabilità del conteggio, occorre attendere (nel caso pessimo) la propagazione della commutazione attraverso gli *n-1* bistabili precedenti. Questo implica un vincolo sulla frequenza ammissibile per il segnale di conteggio una volta che si siano scelte le caratteristiche del bistabile.

Consideriamo ora il caso del contatore decadico per codice BCD: seguendo le stesse regole viste ora, otteniamo la tabella 4.11 (si sottolinea la "circolarità" del conteggio).

Q8	Q4	Q2	Q1	T8	T4	T2	T1
0	0	0	0	0	x2	0	1
0	0	0	1	x5	x1	1	1
0	0	1	0	x4	x0	0	1
0	0	1	1	x3	1	1	1
0	1	0	0	x2	0	0	1
0	1	0	1	x1	x'1	1	1
0	1	1	0	x0	x'0	0	1
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1
1	0	0	1	1	x3	x	1

Tabella 4.11. *Tabella delle verità del asincrono per codice BCD.*

La sintesi ottima richiede ancora l'applicazione dell'impulso di conteggio in AND con T1 (di fatto, essendo T1=1, all'ingresso si applica *t1=Cp*); non occorre invece applicarlo agli altri bistabili, per i quali si riesce sempre a generare autonomamente il fronte di discesa in ingresso. Per gli altri bistabili si ha quindi (rispettando i vincoli sulle condizioni):

t2=T2=Q1

t4=T4=Q2

t8=T8=Q4+Q8Q2

Infine, vale la pena di considerare un ciclo di conteggio del tutto arbitrario per analizzare i problemi che possono porsi nella sintesi di un contatore asincrono. Sia dato il ciclo di conteggio in tabella 4.12 e si vogliano ancora usare, per ovvia

semplicità, bistabili di tipo T.

A	B	C
0	0	0
0	1	1
0	1	0
1	1	0
1	0	1
1	1	1

Tabella 4.12. *Esempio di ciclo di conteggio.*

Cominciamo col compilare la colonna della funzione di eccitazione *Tc*: tentiamo di inserire i fronti di discesa là dove sono necessari, ma troviamo alcune commutazioni *consecutive* che generano 1 consecutivi nella colonna *Tc*: dovremo moltiplicare la funzione *Tc* per *Cp* onde ottenere l'effettivo segnale *tc* applicato in ingresso al bistabile (tabella 4.13). Questo impone di sintetizzare la *Tc come se* si trattasse di un contatore *sincrono*: non possiamo più inserire condizioni d'indifferenza vincolate, poiché la funzione *Tc* sarà in AND con *Cp* e quindi si genererà un 1 ogni volta che a *Tc* viene assegnato il valore 1 (se non si vuole commutazione, *Tc deve* valere 0).

A	B	C	Tc
0	0	0	1
0	0	1	1
0	1	0	0
1	1	0	1
1	0	1	0
1	1	1	1

Tabella 4.13. *Tabella parziale delle eccitazioni del ciclo di conteggio di esempio.*

Risulta:

Tc=A'B'+AB', e tc=Tc.Cp

Allo stesso modo, compiliamo la colonna *Tb*: anche qui vengono richieste tre commutazioni consecutive, quindi *anche la funzione Tb andrà in AND con Cp per generare tb* (tabella 4.14).

A	B	C	Tb	Tc
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
1	1	0	1	1
1	0	1	1	0
1	1	1	1	1

Tabella 4.14. *Ulteriore tabella delle eccitazioni del ciclo di conteggio di esempio.*

Tb=B+C

tb=Tb Cp

Infine, consideriamo la funzione di eccitazione del bistabile *A*. Qui non ci sono commutazioni consecutive; si nota però che la prima commutazione di *A* viene richiesta nel passaggio dalla configurazione 010 a quella 110; ciò significa che nelle due configurazioni consecutive *solo A stesso cambia valore* - gli altri due bistabili mantengono immutata la loro configurazione. Ma ciò significa che, quando è *BC=10*, *non si può generare autonomamente il fronte di discesa* - se manca il segnale di clock, una condizione del tipo "si genera un fronte di discesa su *Ta* se *BC=10* e simultaneamente si genera un fronte di salita su *A* chiaramente non ha senso - il fronte di salita su *A*, a sua volta, non si genera fino a quando non si produce il fronte di discesa su *Ta*. Occorre quindi "forzare" anche qui il fronte di discesa, moltiplicando la configurazione d'ingresso *010* (o un implicante opportunamente ridotto) per il segnale *Cp*. In definitiva, il ciclo di conteggio assegnato *non può essere realizzato mediante un contatore asincrono*; la tabella finale delle eccitazioni è quella in tabella 4.15, e *l'ingresso di conteggio verrà applicato a tutti i bistabili* ottenendo:

Ta=A'B+BC; tb=Ta.Cp

A	B	C	Ta	Tb	Tc
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	0	0
1	1	0	0	1	1
1	0	1	0	1	0
1	1	1	1	1	1

Tabella 4.15. *Tabella delle eccitazioni finale del ciclo di conteggio di esempio.*

4.5 IL PROGETTO BASATO SU MACCHINE A STATI FINITI

La rappresentazione col modello della macchina a stati finiti non è sempre la più utile nella schematizzazione di un sistema dotato di memoria; l'esempio estremo di questo fatto è costituito dalla stessa memoria RAM - evidentemente, una RAM con *n* parole di *k* bit ciascuna *è* una macchina sequenziale e costituisce, a rigore, una macchina a stati finiti con $2^{(kn)}$ stati, ma certamente non è comodo rappresentarla in questo modo. Il modello che si preferisce usare (che suddivide la funzione di indirizzamento e quella di lettura e scrittura) è legato piuttosto alla *funzionalità* del dispositivo, e all'architettura del sistema. Lo stesso vale per molti sistemi che possono essere definiti come *elaborativi*, nei quali cioè esiste una significativa componente con funzioni di *elaborazione dati* eventualmente *controllata* da un'altra componente le cui attività sono facilmente separabili dall'elaborazione. Come esempio, si pensi a un circuito che deve realizzare una moltiplicazione nel modo più banale possibile, cioè per somme ripetute. Se i due fattori hanno lunghezza *k*, il risultato ha potenzialmente lunghezza *2k* e la macchina ha 2^{2k} stati possibili; nessuno sintetizza però questo moltiplicatore come una macchina a stati finiti con 2^{2k} stati, dato che si può schematizzarlo in termini *funzionali* introducendo un *registro moltiplicando* RM, un *registro prodotto* RP di lunghezza *2k*, inizialmente azzerato, un registro *addizionatore*, che riceve ai propri ingressi rispettivamente il moltiplicando e il contenuto del registro prodotto, e un *contatore verso il basso* Cm in cui inizialmente si registra il valore del moltiplicatore. Se il contenuto del contatore Cm è diverso da 0, si somma il contenuto di RP al contenuto di RM, si scrive il risultato nuovamente in RP e si invia un impulso di conteggio a scalare al contatore Cm; quando il contenuto di Cm è pari a 0 non si esegue nessuna somma e il processo ha termine, altrimenti si ripete l'operazione precedente. La struttura risultante realizza la funzionalità richiesta, ma certo non si è proceduto a una

convenzionale sintesi di macchina a stati; di fatto, l'evoluzione del sistema segue uno schema fisso indipendentemente dal particolare stato in cui la macchina si trova - cambia il valore scritto nel registro prodotto e cambia la lunghezza della serie di somme.

4.5.1 La FSM come modello di descrizione di un circuito sequenziale

Il modello a stati si presta invece molto bene a descrivere quei sistemi - o sottosistemi - in cui l'evoluzione della macchina dipende fortemente sia dallo stato presente sia dalla configurazione d'ingresso, e può seguire percorsi nettamente diversi; tratteremo quindi ora l'approccio al progetto di tali sistemi. Innanzitutto, occorre passare dalla descrizione a parole al grafo degli stati; preliminarmente, si controlla che la specifica corrisponda a una macchina *fisicamente realizzabile*, che soddisfi cioè le due seguenti condizioni:

- il numero degli stati è finito, e
- il comportamento della macchina in un istante *t non dipende da eventi futuri*.

Ad esempio, specificare che una macchina S con un ingresso *x* e un'uscita *z* deve fornire uscita 1 "se su *x* si è presentata una sequenza costituita da un numero pari di 0 seguiti da un numero dispari di 1; l'uscita torna a 0 quando il numero di 1 diventa pari (per tornare a 1 se torna dispari) o quando si ripresenta in ingresso uno 0" corrisponde ad una macchina fisicamente realizzabile - come vedremo fra poco; specificare che la macchina deve fornire uscita 1 "se su *x* si è presentata una sequenza costituita da un numero pari di 0 seguiti dal numero dispari di 1 immediatamente superiore " significa indicare una macchina *non fisicamente irrealizzabile* (fatto che si può intuire considerando che si richiede un conteggio di modulo non predeterminato e potenzialmente infinito - quindi, appunto, un numero di stati non finito).

Affrontiamo ora proprio la sintesi della macchina specificata per prima; specifichiamo inoltre che:

- la macchina appartiene al tipo Mealy;
- la macchina ha uno stato iniziale A; finché non si presenta in ingresso il primo 0, non si accetta una sequenza d'ingresso come "utile" (cioè tale da dare uscita 1).

Il secondo punto ci permette di dire che - fino a quando non si presenta *x=0* - la macchina si mantiene nello stato iniziale; l'arrivo di uno 0 in ingresso non genera un 1 in uscita ma porta la macchina in un nuovo stato B (si deve tenere conto del fatto che in ingresso c'è stata una sequenza di 0 di lunghezza dispari) (figura 4.8).

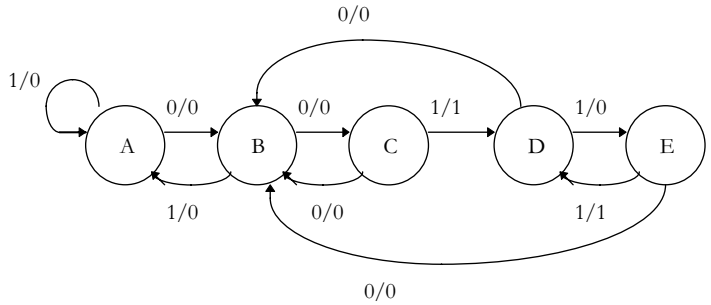


Figura 4.8. *Esempio di FSM.*

Dallo stato B, un ingresso 0 genera una sequenza di 0 di lunghezza pari, corrispondente a un nuovo stato C (la caratteristica che identifica C non era stata trovata precedentemente); l'uscita vale sempre 0. Un ingresso 1, da B, non può corrispondere né a una sequenza utile ne all'inizio di una sequenza utile; ci si riporta nello stato A, che aveva questa caratteristica.

Dallo stato C, un ingresso 0 corrisponde a una sequenza di 0 di lunghezza dispari, cioè alla caratteristica dello stato B, dove si torna - con uscita 0; un ingresso 1 corrisponde a una sequenza di 1 di lunghezza *dispari* che ne segue una di 0 di lunghezza *pari*, quindi si passa a un nuovo stato (D) con uscita che finalmente vale 1. In modo analogo si completa il diagramma, introducendo nuovi stati ogni volta che non si "riconoscono" le caratteristiche di uno stato già esistente, altrimenti tornando a visitare lo stato noto opportuno. La costruzione del grafo termina quando non si introduce più nessun nuovo stato. Ci si può chiedere se non si rischia di fare "esplodere" il numero degli stati inutilmente; come si vedrà più avanti, è sempre possibile *ridurre* tale numero portandosi a una macchina che esibisca il comportamento richiesto col numero minimo degli stati - nella fase iniziale di progetto è meglio introdurre qualche stato "di troppo" piuttosto di commettere errori di progetto ignorando stati che dovrebbero essere previsti.

Dal grafo degli stati si costruisce - senza aggiungere o modificare informazione - la *tabella degli stati* che, come noto, ha tante righe quanti sono gli stati e tante colonne quante sono le configurazioni di valori delle variabili d'ingresso; nel nostro caso, una tabella di cinque righe e due colonne (tabella 4.16)

s^t	*x=0*	*x=1*
A	B/0	A/0
B	C/0	A/0
C	B/0	D/1
D	B/0	E/0
E	B/0	D/1

Tabella 4.16. *Tabella degli stati dell'esempio di figura 4.8.*

Supponiamo per il momento che il progettista non voglia fare interventi di *ottimizzazione* sulla tabella degli stati ottenuta - in particolare, che non voglia tentare di *ridurre* la tabella medesima. I passi successivi richiedono un "aumento di informazione" da parte del progettista; egli deve, innanzitutto, scegliere una *codifica* per gli stati della tabella degli stati - deve compiere cioè quello che si chiama l'*assegnamento* degli stati. A questo scopo, occorre stabilire *il numero minimo di variabili di stato* necessarie per codificare gli stati della macchina: la soluzione è molto semplice - se σ è la cardinalità dell'insieme S degli stati, il minimo numero di variabili di stato è $k=\lceil lg_2\sigma \rceil$. Si tratta poi di associare a ogni stato $s \in S$ una codifica sulle k variabili di stato; nel caso più banale, si può ricorrere ad un'associazione totalmente arbitraria (più avanti si vedrà come una scelta opportuna della codifica possa ridurre la complessità della rete combinatoria). Al termine di questa fase, il progettista può tradurre la *tabella degli stati* in una *tabella delle transizioni*, nella quale per ogni stato presente (codificato) e per ogni configurazione di valori degli ingressi primari si identificano la *codifica* dello stato prossimo e il valore delle uscite. Per l'esempio in esame, la codifica richiede *3* variabili di stato (*y1, y2, y3*) e un assegnamento casuale può essere: *A=000, B=001, C=010, D=011, E=100* La corrispondente tabella delle transizioni è riportata in tabella 4.17.

y1 y2y3	*x=0*	*x=1*
000	001/0	000/0
001	010/0	000/0
010	001/0	011/1
011	001/0	100/0
100	001/0	011/1

Tabella 4.17. *Tabella delle transizioni dell'esempio di figura 4.8.*

Il progettista deve ora identificare *il tipo di bistabile sincrono* che intende usare nel

proprio progetto. La scelta più immediata è quella di bistabili DT; in questo caso infatti lo stato prossimo Y di un bistabile coincide con l'eccitazione D da applicare al bistabile steso, e la tabella delle transizioni può essere immediatamente usata per sintetizzare le funzioni Di=Yi, oltre alle funzioni di uscita. Nel nostro esempio, si ottiene (minimizzando la rete combinatoria):

D1=Y1=xy2y3

D2=Y2=x'y2'y3+xy2y3'+xy1

D3=xy2y3

z= xy2y3'+xy1

Se invece la scelta cade su un altro tipo di bistabile - ad esempio, un bistabile JKT - occorre passare dalla tabella delle transizioni alla *tabella delle eccitazioni*, introducendo (implicitamente) l'informazione costituita dalle funzioni di eccitazione del bistabile scelto. Ripetiamo la sintesi utilizzando appunto bistabili JKT; si ricorda che la tabella delle transizioni di tale tipo di bistabile è quella riportata in tabella 4.18 a destra, partendo dalla tabella delle transizioni di sinistra.

Q	JK=00	01	11	10
0	0	0	1	1
1	1	0	0	1

Q	Q'=0	Q'=1
0	0-	1-
1	-1	-0

Tabella 4.18. *Tabella delle transizioni del flip-flop JKT.*

Ciò porta alla tabella delle eccitazioni in tabella 4.19 (limitatamente alle funzioni di eccitazioni dei tre bistabili; la funzione di uscita, ovviamente, non è influenzata dal tipo di bistabile).

y1 y2y3	***x=0***	***x=1***
000	0-,0-,1-	0-,0-,0-
001	0-,1-,-1	0-,0-,1-
010	0-,-1,1-	0-,-0,1-
011	0-,-1,-0	1-,-1,-1
100	-1,0-,1-	-1,1-,1-

Tabella 4.19. *Tabella delle eccitazioni dell'esempio di figura 4.8.*

e da questa si sintetizzano le sei funzioni J1K1, J2K2, J3K3.

Un'ultima considerazione importante: nell'esempio appena trattato, il numero degli stati non è una potenza di 2; c'è quindi un certo numero di configurazioni di valori delle variabili di stato (nel nostro caso, tre configurazioni) che non vengono associate nella fase di assegnamento a nessuno stato. Nella successiva fase di sintesi

combinatoria tali configurazioni vengono trattate *a tutti gli effetti* come corrispondenti a *condizioni d'indifferenza*, sia per quanto riguarda le funzioni di eccitazione dei bistabili, sia per quanto riguarda la funzione di uscita.

L'esempio ora trattato presuppone l'esistenza di uno *stato iniziale*: mentre l'indicazione di tale stato fa sicuramente parte di quella che si può definire "buona pratica" di progetto, può accadere che non venga data. Il progettista deve allora affrontare prima di tutto il problema di *identificare in modo univoco uno stato* da cui iniziare il proprio diagramma degli stati - e tale identificazione coincide con quella di una *sequenza d'ingresso* (possibilmente breve) che definisca completamente (senza ambiguità) lo stato da cui si comincia a descrivere l'evoluzione della macchina. Consideriamo un caso molto semplice: si vuole progettare una macchina di Mealy con due ingressi *x1, x2* e un'uscita z. L'uscita si porta a 1 se e solo se sui due ingressi si è presentata, sull'arco degli ultimi tre istanti, la sequenza:

x1 x2
0-
-1
10

In tutti gli altri casi è *z=0* (la condizione d'indifferenza significa che è accettabile sia uno 0 sia un 1).

Cominciamo col cercare una sequenza d'ingresso che identifichi completamente uno stato. Possiamo osservare che la configurazione d'ingresso 00 può essere solamente *il primo simbolo di una sequenza utile* - non può comparire in nessun'altra posizione. Tale sequenza "di lunghezza 1" (costituita da un solo simbolo) identifica quindi completamente uno stato A *da cui ha inizio la costruzione del diagramma degli stati* (che tratteremo "come se" fosse uno stato iniziale, sebbene propriamente non lo sia). Caratterizziamo A come lo stato "in cui si è potenzialmente ricevuto il primo simbolo di una sequenza utile".

Se nello stato A si riceve in ingresso il simbolo 00, dato che questo non può che essere il *primo* simbolo di una sequenza utile, la macchina resta nello stato A. Si riceva invece in ingresso il simbolo 10: questo non può essere il *secondo* simbolo di una sequenza utile - ci deve portare in uno stato che indica una sequenza d'ingresso "non riconosciuta" (che non dà uscita 1) e una situazione dalla quale si può "uscire" solo se in ingesso giunge il primo simbolo che *potrebbe* far parte di una sequenza utile. Chiamiamo B lo stato in cui ci si è così portati: data la caratterizzazione di B vediamo che se ora giunge in ingresso un simbolo 10 o un simbolo 11 - simboli che *non* possono dare inizio a una sequenza utile - la situazione non cambia. Si resta quindi in B (con uscita 0). Se invece da B giunge in ingresso un simbolo 00 o 01,

dato che ambedue i simboli possono *dare inizio* a una sequenza utile, ci portiamo in A che è stato caratterizzato in corrispondenza.

Si torni a considerare lo stato A: se in ingresso giunge un simbolo 01, questo *può essere il secondo simbolo di una sequenza utile*; ci portiamo in un nuovo stato C (l'uscita è ancora 0), che deve tener conto anche del fatto che 01, oltre a essere il secondo simbolo di una sequenza utile, può esserne anche il primo. Se da A giunge in ingresso il simbolo 11, questo può *solo* essere il secondo simbolo di una sequenza utile - ci portiamo in un diverso stato D.

Se la macchina è in C, un simbolo 00 riporta in A - 00 può essere solo il primo simbolo di una sequenza utile; 01 *mantiene in B* - la coppia di simboli 01-01 può *globalmente* far parte di una sequenza utile. Il simbolo 11 porta in D - si è creata la sequenza 01-11, potenzialmente parte di una utile (si noti come la "finestra" di potenziale accettazione delle sequenze si sposti lungo la successione di simboli); il simbolo 10 finalmente *conclude una sequenza utile* e porta in uno stato E con uscita 1.

Da D, i simboli d'ingresso 00 e 01 non possono che riportare in A, poiché il simbolo 11 con cui si è giunti in D non può *iniziare* una sequenza utile; il simbolo 11 *ripetuto* due volte non può far parte di una sequenza utile, quindi porta in B. Finalmente, il simbolo 10 conclude anche qui una sequenza utile e porta in E, con uscita 1 (dal momento che E può solo essere il *terzo* simbolo di una sequenza utile - non può trovarvisi in altra posizione - non occorre tenere memoria del particolare simbolo che ha preceduto 10, salvo che della sua accettabilità).

Con E si conclude il diagramma degli stati: i simboli 00 e 01 riportano in A, 10 e 11 portano in B. Il diagramma degli stati è riportato in figura 4.10, la corrispondente tabella degli stati in tabella 4.20.

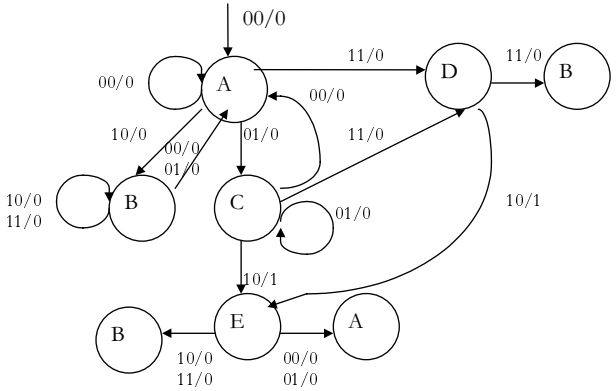


Figura 4.9. *Ulteriore esempio di FSM.*

s^t	x1x2=00	x1x2=01	x1x2=11	x1x2=10
A	A/0	C/0	D/0	B/0
B	A/0	C/0	B/0	B/0
C	A/0	C/0	D/0	E/1
D	A/0	A/0	B/0	E/1
E	A/0	A/0	B/0	B/0

Tabella 4.20. *Tabella degli stati dell'esempio di figura 4.10.*

Infine, sia dato il seguente problema.

Si progetti una *macchina di Moore* con un ingresso *x* e un'uscita *z*; l l'uscita *z* vale 1 se e solo se la sequenza d'ingresso è stata una fra quelle definite da 1-10 (sono cioè considerate utili sia 1110 sia 1010). *Le sequenze utili possono concatenarsi*: ciò vuol dire che, ad esempio, data la sequenza d'ingresso 101010 si avrà uscita 1 sia nello stato raggiunto col quarto bit sia in quella raggiunta col sesto bit.

Anche qui, non viene assegnato a priori uno stato iniziale; si può notare che una sequenza 00 *non* appartiene a nessuna sequenza utile, e può essere usata per definire completamente uno stato da cui iniziare la sintesi. Si è specificato che si vuole sintetizzare una macchina di Moore, quindi l'uscita è associata allo stato. Sia A/0 la coppia stato-uscita raggiunta con la sequenza 00; se in ingresso giungono altri 0 non si modifica la condizione che definisce A, quindi si resta in A. Un ingresso 1 può essere il primo bit di una sequenza utile, quindi porta in un nuovo stato, cui è ancora associata uscita 0 (B/0). Da B, sia ingresso 0 sia ingresso 1 generano sottosequenze di lunghezza due che potrebbero appartenere a una sequenza utile; ci si porta quindi

in due nuovi stati, C e D rispettivamente, ambedue con uscita pari a 0.

Partendo da C, un ingresso 0 ripresenta la sequenza 00 che caratterizza A, quindi riporta in 0; un ingresso 1 corrisponde potenzialmente al terzo bit di una sequenza utile (sottosequenza 101) e porta in uno stato E (con uscita 0). Consideriamo invece D; se l'ingresso vale 0, abbiamo una sottosequenza 110; questa non può nella sua interezza far parte di una sequenza utile, ma gli ultimi due bit - 10 - possono essere i primi di una sequenza utile, corrispondendo alla sottosequenza che ha caratterizzato C; si va quindi in tale stato. Un ingresso 1 genera una sottosequenza 111 potenzialmente parte di una sequenza utile, e porta in un nuovo stato F (sia E sia F hanno uscita 0).

In modo analogo, si vede che da E un ingresso 0 conclude una sequenza utile e porta in uno stato G con uscita 1; un ingresso 1 genera una sequenza 1011 che non può essere utile, ma i cui due ultimi bit possono essere i primi due di una nuova sequenza utile - e porta quindi nello stato D, così caratterizzato. Per quanto riguarda F, l'ingresso 0 conclude una sequenza utile portando in uno stato H con uscita 1, mentre un ingresso 1 genera una sequenza 1111 che non è utile, ma i cui ultimi tre bit possono essere i primi tre di una sequenza utile che ha caratterizzato F, si resta quindi in F.

Infine, da G un ingresso 0 genera una sequenza 00 che caratterizza A; un ingresso 1 genera la sottosequenza 101 che caratterizza E. Da H, l'ingresso porta in A, l'ingresso 1 porta in E (vedi figura 4.10).

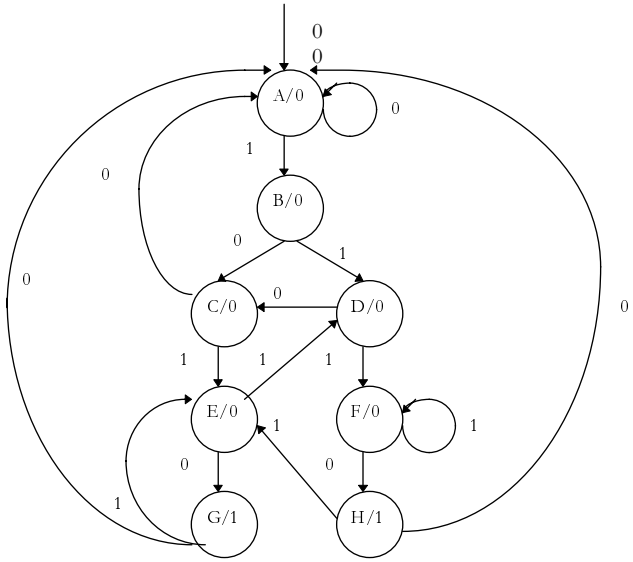


Figura 4.10. *Esempio di FSM di Moore.*

La corrispondente tabella degli stati è riportata in tabella 4.21: dato che si tratta di una macchina di Moore, la tabella comprende una *colonna delle uscite* in cui - per ogni riga (stato presente) si indica il valore dell'uscita, mente nelle caselle all'incrocio stato presente-configurazione degli ingressi primari si riporta solamente lo stato prossimo.

S^t	x=0	x=1	z
A	A	B	0
B	C	D	0
C	A	E	0
D	C	F	0
E	G	D	0
F	H	F	0
G	A	E	1
H	A	E	1

Tabella 4.21. *Tabella degli stati dell'esempio di figura 4.10.*

In assenza di ulteriori fasi di ottimizzazione, la sintesi richiede tre bistabili (gli stati

sono otto). Supponendo poi di fare un assegnamento banale delle tre variabili di stato *y1y2y3*, corrispondente ai numeri d'ordine degli stati nella tabella (da A=000 a H=111), e di usare bistabili DT, la sintesi combinatoria porta a:

z=y1y2

(come era da prevedere in una macchina di Moore, la funzione di uscita dipende solo dalle variabili di stato)

Y1=x'y1y2+xy2+y1y2'y3

Y2=x'y1'y3+y1'y2'y3+x'y1y2'+y1y2'y3'

Y3=xy1'y3+xy1y2'+y1y2'y3

Si vedrà nelle prossime sezioni come procedere alla minimizzazione degli stati e all'assegnamento in modo che anche i risultati della ottimizzazione combinatoria (l'unica qui applicata) vengano giocati nel modo migliore.

4.5.2 Dalla FSM al circuito sequenziale

Esiste uno stretto parallelismo tra FSM e circuiti sequenziali, infatti si può affermare che:

- ogni circuito sequenziale può essere rappresentato da una macchina a stati finiti[2] che, a fronte di simboli di ingresso, che corrispondono a vettori di bit sugli ingressi del circuito, produce dei simboli di uscita che corrispondono ai vettori di bit generati sulle uscite del circuito; e
- ogni macchina a stati finiti può essere trasformata in un circuito sequenziale che, a fronte dei vettori di bit posti sugli ingressi del circuito, corrispondenti ai simboli di ingresso della FSM, genera dei vettori di bit sulle uscite che corrispondono ai simboli di uscita della FSM.

Si tenga presente che non esiste però una corrispondenza biunivoca tra un circuito sequenziale e una FSM (o viceversa) poiché allo stesso circuito possono corrispondere più FSM e la stessa FSM può essere trasformata in più circuiti sequenziali. Esiste quindi una corrispondenza uno a molti tra FSM e circuiti sequenziali e tra circuiti sequenziali e FSM. Le motivazioni di questa corrispondenza sono evidenti dalla fasi della sintesi descritte nel seguito.

2 anche se da un punto di vista pratico il numero di stati potrebbe essere enorme.

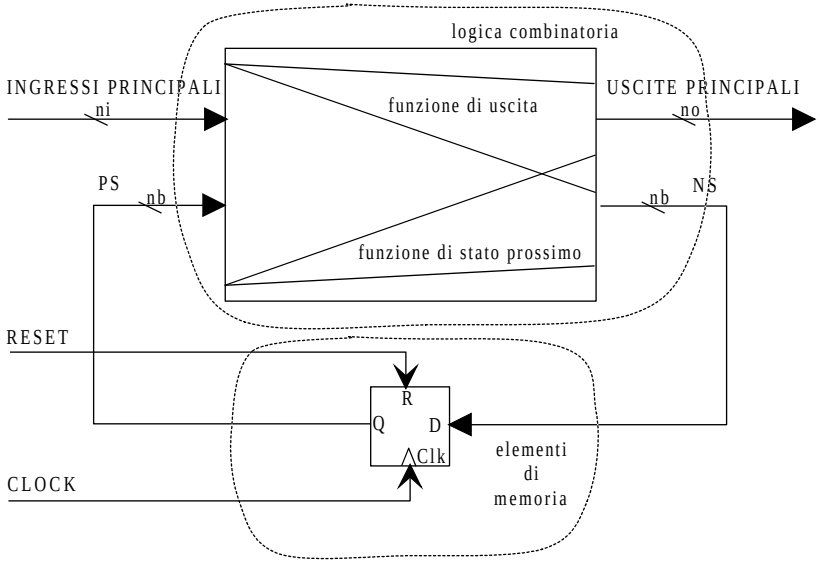


Figura 4.11. *Modello di Huffman di un circuito sequenziale.*

Si definisce **sintesi** di una FSM l'insieme delle fasi necessarie a trasformare una FSM in un circuito sequenziale che la realizza. Questa trasformazione *non ottimizzata* è già stata presentata nei corsi di base sulla progettazione di circuiti digitali. Si basa sul modello di *Huffman*, riportato in figura 11, che permette di rappresentare qualsiasi circuito sequenziale, ma, isolando gli elementi di memoria dalla logica combinatoria, è particolarmente adatto a descrivere la relazione tra macchine a stati finiti e circuiti sequenziali.

Si può facilmente osservare che:

- assegnata una codifica su *nb* bit agli stati della FSM, gli *nb* elementi di memoria ne permettono la memorizzare;
- questi elementi di memoria caricano la codifica degli stati prossimi attraverso le linee di stato prossimo NS_0 ... NS_{nb-1} e generano le codifiche degli stati presenti sulle linee di stato presente PS_0 ... PS_{nb-1};
- la linea di ingresso RESET forza la configurazione di reset negli elementi di memoria; ossia, forza il caricamento dello stato di reset *s*$_r$;
- su ogni nuovo fronte di salita del segnale CLOCK, la logica combinatoria calcola i nuovi valori delle uscite e delle codifiche degli stati prossimi, ossia implementa le funzioni λ e δ.

La sintesi ottima di una FSM si basa invece sulle fasi già informalmente presentate in

questo capitolo, ma che vengono riprese e approfondite nel seguito. Il termine sintesi ottima è qui usato in maniera impropria poiché la sintesi ottima di un circuito sequenziale, come già di un circuito combinatorio, è un problema NP-completo che non può quindi essere affrontato con algoritmi esatti, ma che richiede l'utilizzo di tecniche euristiche. Il raggiungimento dell'ottimo assoluto non è quindi garantito, ma tutti gli algoritmi utilizzati convergono ad un ottimo locale che rappresenta il meglio che si può ottenere in tempi ragionevoli.

I passi della sintesi possono essere così riassunti.

- Identificazione di una descrizione **minima** del circuito sequenziale da realizzare. Il paragrafo 4.6 definisce il concetto di *equivalenza* (o compatibilità) di due FSM e propone un algoritmo per l'identificazione di una FSM composta da un numero di stati minimo, ma equivalente alla FSM data. La minimizzazione del numero degli stati permette generalmente di ridurre le dimensioni del circuito sequenziale che realizza la FSM.
- Assegnamento di una **codifica degli stati** che minimizza una funzione obiettivo (paragrafo 4.7). Il criterio di minimizzazione qui presentato è quello dell'area, altri criteri possono riguardare la velocità del circuito prodotto, la dissipazione di potenza, la collaudabilità, ecc. In ogni caso, le codifiche assegnate agli stati hanno un impatto notevole sulle caratteristiche del circuito realizzato.
- **Mimimizzazione logica** del circuito combinatorio che realizza le funzioni λ e δ. La presenza degli elementi di memoria permette di estendere l'applicazione degli algoritmi di minimizzazione logica utilizzati per la sintesi di circuiti combinatori presentati nel capitolo 2.
- **Mapping tecnologico**. Le equazioni generiche che descrivono i nodi della rete vengono realizzate mediante componenti fisici effettivi appartenenti ad una libreria tecnologica (nel caso di realizzazioni *semi* o *full custom*) o mediante le LUT di una FPGA. Questa fase è equivalente alla corrispondente già vista per la realizzazione di circuiti combinatori e presentata nel capitolo 3.

Si esaminano ora le differenti fasi.

4.6 LA MINIMIZZAZIONE DEGLI STATI

Minimizzare il numero degli stati permette in genere di ridurre la complessità del circuito sequenziale che realizza la FSM, perché:

- si può ridurre il numero di variabili di stato e quindi di bistabili necessario (ovviamente, la riduzione dipende secondo una relazione logaritmica dal numero degli stati; se si passa da 7 a 6 stati, il numero dei bistabili necessari resta 3!);

- dato che le due funzioni λ e δ dipendono dalle variabili di stato, se si riduce il numero di queste ultime in genere si riduce anche la complessità delle funzioni;
- se si aumenta il numero di configurazioni delle variabili di stato non utilizzate per la codifica, aumentano le probabilità di ridurre la complessità della rete combinatoria.

Considereremo innanzitutto il caso delle macchine *completamente specificate*, quelle cioè in cui per ogni coppia stato presente-configurazione d'ingresso risultano specificati sia stato prossimo sia configurazione di uscita (ciò accade, in particolare, negli esempi visti finora).

4.6.1 Le macchine completamente specificate

Date due macchine a stati finiti $M1=<S_1,I_1,O_1,\lambda_1,\delta_1>$ e $M2=<S_2,I_2,O_2,\lambda_2,\delta_2>$ tali che sia $I_1=I_2$ e $O_1=O_2$, potremo dire che le due macchine sono *equivalenti* se e solo se si verifica la seguente condizione:

- per ogni stato $s_1^a \in S_1$ esiste uno stato $s_2^b \in S_2$ tale che, ponendo la macchina *M1* in s_1^a e la macchina *M2* in s_2^b e applicando alle due macchine una qualsiasi sequenza d'ingresso I_α, le due sequenze di uscita prodotte risultino a loro volta identiche. La proprietà deve valere anche per *M2* rispetto a *M1*.

La definizione di equivalenza risulta semplificata se ambedue le macchine sono dotate di uno stato iniziale (o "di reset"), rispettivamente, s_{r1} e s_{r2}; in questo caso le macchine sono definibili come $M1=<S_1,I_1,O_1,\lambda_1,\delta_1, s_{r1}>$ e $M2=<S_2,I_2,O_2,\lambda_2,\delta_2, s_{r2}>$ e la definizione di equivalenza diventa:

- per ogni sequenza d'ingresso I_α, ponendo le due macchine nei rispettivi stati iniziali si generano identiche sequenze di uscita.

Si noti come la definizione di equivalenza, in ambedue i casi, faccia riferimento solamente alle relazioni ingresso-uscita; è quindi possibile che le due macchine abbiano *insiemi di stati diversi* - in particolare, con *diverse cardinalità*.

Consideriamo ora una macchina $M=<S,I,O,\lambda,\delta>$ completamente specificata. Diremo che due stati $s_i,s_k \in S$ sono *equivalenti o* indistinguibili (relazione indicata con $s_i \sim s_k$) se e solo se si verifica che per una qualsiasi sequenza d'ingresso $I_\alpha=i1\ldots ik$ le sequenze di uscita prodotte, ponendo la macchina inizialmente in s_i oppure in s_k, sono identiche (formalmente: $\lambda(s_i, I_\alpha) = \lambda(s_k, I_\alpha), \forall I_\alpha$). Si verifica immediatamente - per la stessa definizione che se ne è data - che la relazione di indistinguibilità è:

- riflessiva - $s_i \sim s_i$;
- simmetrica - se $s_i \sim s_k$, è anche $s_k \sim s_i$;
- transitiva - se $s_i \sim s_k$ e $s_k \sim s_j$, è anche $s_i \sim s_j$.

Di conseguenza, la relazione di indisitinguibilità è una *relazione di equivalenza*, e come tale *induce sull'insieme degli stati della macchina M una partizione di equivalenza* π*=<B1,…,Br>* tale che due stati s_i ed s_k possono appartenere ad uno stesso blocco *Bm* se e solo se $s_i \sim s_k$, mentre i due stati appartengono a blocchi diversi se e solo se s_i *non* è equivalente a s_k.

Diremo che una macchina *M* è *minima* se non esiste nel suo insieme di stati *S* nessuna coppia di stati s_i,s_k tali che $s_i \sim s_k$.

Il problema della minimizzazione degli stati di una data macchina *M* può essere riformulato come quello dell'identificazione della *minima macchina equivalente ad M*; a questo scopo, si supponga di avere identificato la partizione di equivalenza π*=<B1,…,Br>* indotta sull'insieme degli stati *S* di *M* dalla relazione di indistinguibilità.

È immediatamente dimostrabile che una macchina *M'* equivalente alla macchina *M* e con un insieme di stati *T:(t1..tr)* tali che ogni stato *ti* risulti equivalente agli stati del blocco *Bi* sarà la macchina *minima* fra quelle equivalenti a *M*; si noti che, date le caratteristiche della relazione di equivalenza, la macchina minima è *unica*. Il problema evidentemente è quello di determinare la partizione di equivalenza; la definizione di indistinguibilità ci porterebbe potenzialmente ad esaminare le risposte della macchina, partendo dai vari stati, ad un insieme infinito di sequenze di lunghezza infinita - evidentemente, non un metodo applicabile. Si tratta quindi di presentare un metodo praticamente applicabile per determinare la partizione di equivalenza su *S*: di fatto, grazie alla proprietà transitiva, basta determinare tutte le possibili relazioni di indistinguibilità fra *coppie di stati*.

A questo scopo, è utilizzato l'algoritmo di *Paull-Unger*, che si basa sulla seguente ridefinizione dell'indistinguibilità fra due stati $s_i,s_j \in S$:

si~sj se e solo se, $\forall i_a \in I$*, si verificano le seguenti due condizioni*

- $\lambda(s_i, i_a) = \lambda(s_j, i_a)$
- $\delta(s_i, i_a) \sim \delta(s_j, i_a)$

(la modifica per macchine di Moore è ovvia). Si opera ora su sequenze di un solo simbolo, ma si è introdotta una condizione *ricorsiva*: la seconda condizione implica infatti una nuova verifica di indistinguibilità sulla coppia di stati prossimi. Poiché l'insieme di stati *S* è per definizione finito, dopo un numero finito di confronti (al massimo, pari al numero di tutte le possibili coppie di stati) si verificherà una fra le due seguenti situazioni:

1) $s_i \neg\sim s_j$ se

- le uscite sono diverse ($\exists i_b \in I$ tale che $\lambda(s_i, i_b) \neq \lambda(s_j, i_b)$), oppure

- $\exists i_b \in I$ tale che $\delta(s_i, i_a) \neg\sim \delta(s_j, i_a)$(si sono raggiunti due stati prossimi di cui è stata già provata la *distinguibilità*)

2) $s_i \sim s_j$ se $\forall i_a \in I$ è $\delta(s_i, i_a) \sim \delta(s_j, i_a)$ cioè se si è già dimostrato che gli stati prossimi sono equivalenti.

Un caso particolare si trova quando la condizione è $\delta(s_i, i_a)=si$ e $\delta(s_j, i_a) =sj$ o $\delta(s_i, i_a)=sj$ e $\delta(s_j, i_a) =s_i$, quando cioè l'equivalenza di una coppia di stati è vincolata dall'equivalenza della *stessa* coppia di stati: si ha allora un caso di immediata "circolarità di vincolo" che è soddisfatta a priori per tautologia.

Il metodo di Paull-Unger implica il confronto fra tutte le coppie di stati in *S*; dato che la relazione di indistinguibilità è simmetrica e riflessiva, se σ è la cardinalità di S non è necessario fare σ^2 confronti fra coppie di stati ma ne sono sufficienti $\frac{\sigma^2 - \sigma}{2}$. Per operarli, si costruisce una tabella triangolare, detta *delle implicazioni*, in cui ogni coppia di stati $s_i,s_j / i \neq j$ compare una e una sola volta: nella casella corrispondente alla coppia s_i,s_j si inserisce, in una prima passata basata sul semplice esame della tabella degli stati:

- il simbolo ~ se si verifica immediatamente la *indistinguibilità* dei due stati;
- il simbolo X se si verifica immediatamente la *distinguibilità* dei due stati;
- l'insieme di condizioni (s_k,s_h) cui è vincolata $s_i \sim s_j$ se indistinguibilità o distinguibilità non sono dimostrabili in questo passo.

Compiuta questa analisi, se in tutte le celle compare il simbolo di equivalenza o quello di distinguibilità l'algoritmo termina; altrimenti, si analizzano tutte le condizioni di cui al punto c). Se nella cella corrispondente agli stati s_i,s_j si identifica *almeno una condizione* non soddisfatta - se cioè per almeno una coppia di stati prossimi s_h,s_k è $s_h \neg\sim s_k$, è anche $s_i \neg\sim s_j$; se *per tutte* le coppie di stati prossimi è $s_h \sim s_k$, allora è anche $s_i \sim s_j$; se non è possibile il controllo, si rinvia ad un passo successivo dell'algoritmo.

Dopo un numero finito di passi, ci troviamo in una delle seguenti condizioni:

- tutte le coppie di stati sono state classificate come distinguibili oppure indistinguibili;
- si presenta un caso di *circolarità di vincolo* del tipo $s_i \sim s_j \rightarrow s_h \sim s_k \rightarrow s_m \sim s_p \ldots \rightarrow s_i \sim s_j$. In questo caso si può concludere che la relazione di indistinguibilità è soddisfatta *per tutte le coppie incluse nell'anello* - è cioè $s_i \sim s_j$, $s_h \sim s_k$, $s_m \sim s_p$.

Applichiamo questo algoritmo innanzitutto alla tabella 4.16, relativa all'esempio di figura 4.8, che riportiamo qui per comodità:

s^t	*x=0*	*x=1*
A	B/0	A/0
B	C/0	A/0
C	B/0	D/1
D	B/0	E/0
E	B/0	D/1

La tabella delle implicazioni, costruita secondo le regole ora viste, è mostrata in tabella 4.22.

B	B,C			
C	X	X		
D	A,E	A,E	X	
E	X	X	~	X
	A	**B**	**C**	**D**

Tabella 4.22. *Tabella delle implicazioni iniziale dell'esempio di figura 4.8.*

Nella seconda passata, si verifica che le due condizioni - che implicherebbero rispettivamente B~C e A~E, *non* sono soddisfatte; quindi la tabella viene modificata come in tabella 4.23.

B	X			
C	X	X		
D	X	X	X	
E	X	X	~	X
	A	**B**	**C**	**D**

Tabella 4.23. *Tabella delle implicazioni finale dell'esempio di figura 4.8.*

La partizione di equivalenza ottenuta è quindi $\pi = (A, B, CE, D)$. La macchina minima equivalente a quella data, per quanto già detto, sarà dunque una macchina con *quattro* stati, corrispondenti ognuno a un blocco della partizione di equivalenza: vediamo come costruire tale macchina.

Indichiamo i blocchi della partizione di equivalenza (e quindi gli stati della macchina minima) come α=A, β=B, γ=C E, δ=D. Per ogni stato della macchina minima, e per ogni simbolo d'ingresso, le funzioni stato prossimo e uscita vengono definite come segue:

- lo stato prossimo corrisponde al blocco della partizione di equivalenza cui appartengono gli stati prossimi degli stati appartenenti al blocco "di partenza";
- l'uscita è identica a quella di tutti gli stati appartenenti al blocco di partenza.

Con queste regole, si costruisce la tabella degli stati riportata in tabella 4.24.

s^t	*x=0*	*x=1*
α	β/0	α/0
β	γ/0	α/0
γ	β/0	δ/1
δ	β/0	γ/0

Tabella 4.24. *Tabella degli stati minima dell'esempio di figura 4.8.*

Si nota subito come ora la codifica sia possibile con solo *due* variabili di stato; la rete finale sarà quindi certamente meno costosa di quella inizialmente progettata.

Si consideri ora la FSM descritta mediante la tabella 4.25.

	0	**1**
A	G / 00	C / 01
B	G / 00	D / 01
C	D / 10	A / 11
D	C / 10	B / 11
E	G / 00	F / 01
F	F / 10	E / 11
G	A / 01	F / 11

Tabella 4.25. *Tabella degli stati di un esempio di FSM.*

La FSM è composta da 7 stati, quindi la tabella delle implicazioni è composta da $\frac{\sigma^2 - \sigma}{2} = \frac{7^2 - 7}{2} = 21$ celle, che permettono di effettuare i seguenti confronti tra coppie di stati.

Il *primo passo* dell'algoritmo esamina tutte le celle ed esegue le classificazioni presentate in tabella 4.26.

	A	B	C	D	E	F
B	(C, D)					
C	×	×				
D	×	×	(A, B)			
E	(C, F)	(D, F)	×	×		
F	×	×	(A, E) (D, F)	(B, E) (C, F)	×	
G	×	×	×	×	×	×

Tabella 4.26. *Tabella delle implicazioni iniziale dell'esempio di tabella 4.25.*

La prima cella, relativa al confronto tra *A* e *B*, rimanda il confronto alla relazione (non ancora identificata) tra gli stati *C* e *D*. Infatti, a fronte del simbolo di ingresso 0, sia *A* che *B* generano la stessa uscita e si portano nello stesso stato prossimo, mentre a fronte dell'ingresso 1, generano la stessa uscita, ma si portano rispettivamente nello stato *C* e *D*. L'equivalenza di *A* e *B* dipende quindi dall'equivalenza di *C* e *D* che deve essere dimostrata nei passi successivi dell'algoritmo.

Il *secondo passo* dell'algoritmo riesamina tutte le coppie di stati non ancora classificate, ma non riesce a compiere alcuna ulteriore classificazione. Si è quindi di fronte a delle relazioni circolari del tipo: A ~ *B* se *C~D*, ma *C~D* se *A~B* per cui si può concludere che *A~B* e *C~D*. La partizione di equivalenza può essere evidenziata in forma grafica con un *grafo delle equivalenze* (figura 6.12) in cui ogni nodo rappresenta uno stato e due nodi sono collegati da un lato se e solo se i due corrispondenti stati risultano equivalenti:

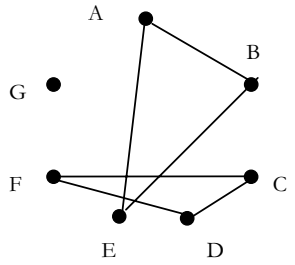


Figura 4.12. *Grafo di equivalenza della tabella 4.26.*

Si può vedere che i *blocchi* della partizione di equivalenza corrispondono nel grafo a *poligoni completi*, nei quali cioè ogni vertice è collegato a ogni altro vertice del nodo (il che corrisponde alla definizione del blocco di equivalenza, nel quale ogni stato è equivalente a ogni altro stato del blocco stesso): tali poligoni sono mutuamente

disgiunti. I blocchi della partizione di equivalenza sono:

1={A,B,E}

2={C,D,F},

3={G}.

Associando un solo stato ad ogni blocco della partizione si ottiene la FSM minima equivalente rappresentata dalla tabella delle transizioni 4.27.

	0	**1**
1	3 / 00	2 / 01
2	2/ 10	1 / 11
3	1 / 01	2 / 11

Tabella 4.27. *Tabella delle transizioni della FSM minima equivalente alla FSM di tabella 4.25.*

In questo caso il vantaggio di realizzare il circuito sequenziale relativo alla FSM minima equivalente, invece che il circuito relativo alla FSM iniziale, è misurabile dal numero di elementi di memoria necessari a codificare gli stati che passano da 3 (per sette stati) a 2 (per 3 stati) e dalle dimensioni delle funzioni di stato prossimo e di uscita che riguardano 6 transizioni invece che 14.

4.6.2 Le macchine incompletamente specificate

Nel caso visto ora di una FSM completamente specificata, è possibile identificare in maniera esatta la FSM minima equivalente utilizzando un algoritmo (come quello di Paull-Unger) che ha una complessità esponenziale con il numero di stati della FSM e che genera la macchina minima *unica* grazie al fatto che anche la partizione di equivalenza è **unica**.

Si possono però presentare tabelle degli stati di FSM solo parzialmente specificata: si pensi ad una macchina i cui ingressi codificano le cifre decimali in codice BCD - delle sedici configurazioni d'ingresso possibili solo dieci vengono utilizzate. In corrispondenza delle altre, *che non possono darsi*, stato prossimo e uscita saranno "condizioni di indifferenza": più in generale, è possibile trovarsi di fronte a tabelle degli stati per le quali, in corrispondenza di particolari configurazioni ingressi primari/stato presente o il simbolo d'uscita, o lo stato prossimo, o ambedue non sono specificati. Volendo affrontare la minimizzazione di tali macchine il problema si complica notevolmente. Dalla definizione ricorsiva precedentemente, due stati s_j e s_k sono indistinguibili se $\forall\ i_i \in I$:

- $\lambda(s_j, i_i) = \lambda(s_k, i_i)$ e
- $\delta(s_j, i_i) \sim \delta(s_k, i_i)$.

Si può invece definire che in una FSM parzialmente specificata due stati s_j e s_k sono **compatibili** (s_j ν s_k) se ∀ $i_i \in I$:

- ovunque siano ambedue specificati, è $\delta(s_i,i_i)\vee\delta(s_j,i_i)$, e
- ovunque siano ambedue specificate, è $\lambda(s_i,i_i)\vee\lambda(s_j,i_i)$,

(la definizione si modifica in modo ovvio per le macchine di Moore). Si noti come un diverso assegnamento dei valori non specificati possa far sì che due stati siano o non siano compatibili. La relazione di compatibilità *non ammette la proprietà transitiva* dato che se s_j ν s_k e s_k ν s_i ***non*** è detto che s_j ν s_i . Infatti, gli assegnamenti dei bit di uscita non specificati, o degli stati non specificati, che permettono di affermare che s_j ν s_k , potrebbero essere in contrasto con gli assegnamenti che permettono di affermare che s_k ν s_i . Ne segue che la relazione di compatibilità *non è una relazione di equivalenza*.

Si consideri per esempio la FSM descritta dalla tabella degli stati 4.28.

	0	**1**
A	E / 0	A / 0
B	D / 0	B / 0
C	E / -	C / -
D	A / 1	A / 1
E	A / -	B / -

Tabella 4.28. *Tabella degli stati di una FSM parzialmente specificata.*

Si costruisca ora la tabella delle implicazioni e si esegua il primo passo dell'algoritmo di Paull-Unger (tabella 4.29): finché si confrontano transizioni con uscite completamente specificate (per esempio durante il confronto tra *A* e *B*) l'algoritmo non cambia. Altrimenti, si introduce il *vincolo* di "potenziale compatibilità" che dipende in sostanza da un possibile assegnamento delle uscite non specificate; il fatto che la relazione di compatibilità non sia transitiva induce a non potere immediatamente cancellare i vincoli ogni volta che essi non siano violati e, come si farebbe nel caso di macchine completamente specificate, poiché possono esserci vincoli non violati ma contraddittori.

B	(D, E)			
C	ν	(D, E)		
D	×	×	(A, E) (A,,C)	
E	(A, B)	×	(A, E) (B, C)	(A, B)
	A	**B**	**C**	**D**

Tabella 4.29. *Tabella delle implicazioni iniziale dell'esempio di tabella 4.28.*

Si noti che nei passi successivi dell'algoritmo si deve tener conto di queste ipotesi altrimenti si possono generare dei conflitti tra gli assegnamenti dei bit non specificati. Per esempio, nel confronto tra *C* e *D* si rimanda l'analisi di compatibilità al confronto tra *A* e *E* e tra *A* e *C* facendo l'ipotesi che le uscite non specificate delle transizioni uscenti da *C* siano poste a 1, altrimenti *C* sarebbe sicuramente non compatibile con *D*. Se si stesse ricercando l'indistinguibilità tra gli stati, non sarebbe necessario tener conto anche del confronto tra *A* e *C* poiché questi due stati (si veda la casella corrispondente) sono già stati dichiarati compatibili. Si noti però che $A\vee C$ perché le transizioni uscenti da *C* sono state poste a 1, condizione che è incompatibile con la condizione appena espressa. In conclusione, è necessario mantenere nelle caselle della tabella delle implicazioni tutti i vincoli in modo da poterne verificare la congruenza alla fine dell'algoritmo. Questa congruenza viene esaminata costruendo le classi di compatibilità (così come sono state identificate le classi, o blocchi, di equivalenza): tali classi o non costituiscono però una partizione - vale a dire che uno stesso stato può appartenere a più classi di compatibilità diverse, cui corrispondono risoluzioni diverse dei vincoli (assegnamenti diversi delle condizioni non specificate). Occorre quindi verificare tutte le possibili alternative che si hanno a disposizione specificando i vincoli sulla compatibilità tra coppie di stati perché la classe possa esistere. Per esempio,

- la compatibilità fra *A* e *B* è condizionata a quella fra *D* ed *E*.

Per descrivere tutti questi vincoli è utile costruire un **grafo di compatibilità** analogo a quello delle equivalenze già visto; qui però un arco fra due nodi può essere *incondizionato* (la compatibilità è comunque soddisfatta indipendentemente dalle scelte per le condizioni non specificate) oppure contrassegnato da una *condizione* corrispondente alla compatibilità fra altre coppie di stati. In questo secondo caso, la compatibilità esiste se e solo se le compatibilità corrispondenti a tutti i vincoli vengono *accettate e utilizzate* - in altre parole, se nella macchina minima finale ogni coppia di stati corrispondente a un vincolo verrà fusa in un unico stato. Per l'esempio in esame si ottiene il grafo in figura 4.13.

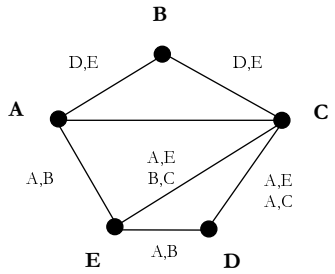


Figura 4.13. *Grafo di compatibilità della tabella 4.29.*

Una classe di compatibilità è costituita da un insieme di stati che sono tutti due a due compatibili fra di loro - è quindi ancora rappresentata sul grafo da un poligono completo; una **classe massima di compatibilità** (classe di dimensione massima non contenuta in un'altra classe) è rappresentata da un poligono completo non contenuto in nessun altro poligono. Mentre però nel caso dell'equivalenza i poligoni completi massimi erano tutti disgiunti fra di loro, qui questo fatto non è più verificato: le classi massime di compatibilità sono:

- α=*A,B,C* condizionata dall'assegnare D ed E a una stessa classe di compatibilità realizzata poi con un solo stato nella macchina minima;
- β=*A,C,E*, condizionata alla scelta di una classe di compatibilità che includa A e B, una che includa A ed E e una che includa B e C
- γ=*D,E*, condizionata alla scelta di una classe di compatibilità che includa A e B.

Poiché le classi di compatibilità **non** sono disgiunte, non è più possibile realizzare la macchina minima associando semplicemente uno stato a ogni classe massima di compatibilità: in effetti, è stato dimostato che *il numero di classi massime di compatibilità costituisce un limite superiore per il numero di stati della macchina minima*. Inoltre, data la libertà di assegnamento per le condizioni non specificate, in genere *la macchina minima non è unica*: anche se (ovviamente) il numero degli stati non cambia, possono cambiare le transizioni fra stati e gli assegnamenti delle condizioni non specificate. La mancanza di disgiunzione non permette di risolvere il problema della minimizzazione in forma esatta, ma obbliga ad utilizzare un algoritmo euristico per la ricerca di una buona soluzione. Sono stati proposti svariati algoritmi per identificare una soluzione, ma la loro trattazione esula dagli argomenti di questo libro. Un semplice algoritmo *greedy* per la soluzione del problema è il seguente:

1) si sceglie una classe massima di compatibilità, cui si associa uno stato della macchina minimale;
2) si rimuovono dal grafo di compatibilità tutti i nodi di questa classe (ed i relativi archi) se non implicano la violazione di vincoli di compatibilità,
3) se il grafo delle compatibilità contiene ancora dei poligoni, si torna al punto 1.

Applicando questo di algoritmo e scegliendo come classe di partenza α, si trova la seguente soluzione:

- α=*A,B,C*,
- β=*D,E*,

Passando ora a costruire la macchina corrispondente, per ogni stato t_i della macchina minimale corrispondente a una classe di compatibilità C_i della macchina iniziale e per ogni simbolo d'ingresso *i** si applicano le seguenti regole:

1) se per almeno uno degli stati appartenenti alla classe di compatibilità C_i lo stato prossimo è specificato, nella macchina minimale lo stato prossimo di t_i è quello associato alla classe di compatibilità che include gli stati prossimi di C_i ;
2) se per almeno uno degli stati appartenenti alla classe di compatibilità C_i l'uscita è specificata, nella macchina minimale l'uscita associata a t_i assume tale valore;
3) in ogni altro caso, si mantengono le condizioni non specificate.

In corispondenza, si ricava la seguente FSM compatibile con quella iniziale:

	0	**1**
α	β / 0	α / 0
β	α / 1	α / 1

Questa soluzione corrisponde ad aver scelto di completare i bit di uscita delle transizioni uscenti da *C* con 0, poiché compatibile con *A* e *B*, e di quelle uscenti da *E* con 1, poiché compatibile con *D*.

Si noti infine che la scelta di tutte le classi di massima compatibilità avrebbe identificato una FSM con tre stati, ossia di dimensioni superiori.

Si consideri ora la macchina definita dalla tabella degli stati 4.30.

	00	01	11	10
A	B,0	C,-	--	D,0
B	B,0	--	E,1	D,0
C	F,1	A,0	B,-	E,1
D	F,-	E,1	--	A,0
E	--	E,1	E,1	D,0
F	--	B,0	B,0	--

Tabella 4.30. *Tabella degli stati di una FSM parzialmente specificata.*

La tabella delle compatibilità è riportata in tabella 4.31.

B	ν				
C	×	×			
D	(B,F) (C,E)	(B,F) (A,D)	×		
E	(C,E)	ν	×	(A,D)	
F	(B,C)	×	(A,B)	×	×
	A	**B**	**C**	**D**	E

Tabella 4.31. *Tabella delle implicazioni iniziale dell'esempio di tabella 4.30.*

L'analisi di questa tabella porta a stabilire che:

- essendo B,F incompatibili, anche A e D lo sono, come B e D; analogamente, l'incompatibilità di B e C porta a stabilire quella fra A ed F, e quella fra C ed E porta all'incompatibilità fra A ed E;
- dal passo precedente, si deduce l'incompatibilità fra D ed E.

La tabella delle implicazioni aggiornata è riportata in tabella 4.32

B	ν				
C	×	×			
D	×	×	×		
E	×	ν	×	×	
F	×	×	A,B	×	×
	A	**B**	**C**	**D**	E

Tabella 4.32. *Tabella delle implicazioni finale dell'esempio di tabella 4.30.*

A questa tabella corrisponde il grafo delle compatibilità in figura 4.14:

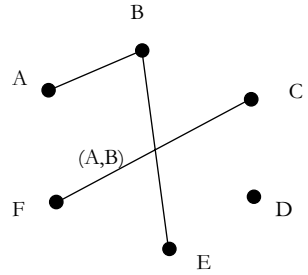


Figura 4.14. *Grafo di compatibilità della tabella 4.32.*

Le classi massime di compatibilità sono:

1 = A,B

2 = B,E

3 = C,F

4 = D

La macchina minima avrà quindi al più quattro stati. Si noti che, scegliendo tutte le classi massime di compatibilità, le condizioni sono implicitamente tutte soddisfatte; si hanno però, anche in questo esempio, classi *non disgiunte*; lo stato B appartiene sia alla classe 1 sia a quella 2 - al momento della realizzazione della macchina minima, occorrerà stabilire per ogni comparsa dello stato prossimo B se nella macchina minima ad esso corrisponde 1 piuttosto che 2. La scelta delle classi massime porta alla tabella 4.33.

	00	01	11	10
1	1,0	3,-	4,1	2,0
2	2,0	2,1	2,1	4,0
3	3,1	1,0	1,0	2,1
4	3,-	2,1	--	1,0

Tabella 4.33. *Tabella degli stati della FSM minima compatibile alla FSM di tabella 4.30.*

Esaminando la costruzione di questa tabella, si può notare che:

- la riga 1 deriva dagli stati A e B; in colonna 00 lo stato prossimo era per i due stati originali sempre B - si può porre come stato prossimo uno qualsiasi degli stati della macchina minima che includono B (per esempio, lo stesso stato 1); in colonna 01, solo per lo stato A si ha stato prossimo specificato (non l'uscita): *la condizione specificata prevale sempre*, quindi si pone come stato prossimo 3 (che include lo stato C); analogamente, in colonna 11 si specificano stato prossimo e uscita come derivanti dallo stato B (stato prossimo 2, uscita 1);
- la riga 2 deriva dagli stati B ed E; nella colonna 00 solo per B si ha stato prossimo specificato, quindi dallo stato 2 si può porre come stato prossimo sia 1 sia 2 - qui si è arbitrariamente scelto 2; e così via, fino al completamento della tabella.

Nella fase di minimizzazione, si può anche scegliere (come già detto) un insieme di classi di compatibilità *non necessariamente (tutte) massime*, a patto di rispettare i vincoli posti dalla tabella delle compatibilità e di non aumentare il numero degli stati. Nel nostro caso, la scelta delle classi (A,B), (C,F), (D), (E) avrebbe portato ancora a una macchina con soli quattro stati rispettando il vincolo imposto, per la validità della classe (C,F), dall'avere attribuito A e B a una stessa classe di compatibilità; fra l'altro, la tabella finale sarebbe stata meno specificata di quella vista ora, e quindi con migliori probabilità di minimizzazione combinatoria. La scelta (A), (B,E), (C,F), (D) non sarebbe invece stata possibile, dato che non avrebbe rispettato il vincolo che vuole A e B nella stessa classe per garantire la compatibilità tra C e F.

4.7 L'ASSEGNAMENTO DEGLI STATI

La codifica degli stati ha un notevole impatto sulle dimensioni e prestazioni dei circuiti realizzati. Un assegnamento casuale, come quello visto finora, porta generalmente ad una soluzione lontana da una soluzione ottima - e questo è tanto più

probabile quanto maggiore è il numero degli stati. Peraltro, una ricerca esaustiva nello spazio delle possibili codifiche (esaminando il risultato di una sintesi combinatoria per ogni possibile alternativa di codifica degli stati) non è perseguibile se non per numero di stati ridicolmente basso, visto l'ampiezza di questo spazio. Se infatti ci si concentra anche solamente sulle codifiche a lunghezza minima, il numero di elementi di memoria necessari a codificare l'insieme degli stati *S* è:

$$nb = \lceil log_2|S| \rceil$$

dove il simbolo $\lceil\ \rceil$ indica l'intero superiore. Esistono quindi 2^{nb} codifiche che possono essere assegnate agli $|S|$ stati in:

$$\frac{(2^{nb}-1)!}{(2^{nb}-|S|)!\ nb!}$$

modi diversi. Un'analisi esaustiva del problema risulta improponibile,; si devono quindi identificare degli efficaci criteri euristici.

Per effettuare un'analisi dell'efficacia dei criteri, restringiamo il campo della codifica degli stati al seguente caso particolare:

- si considerano codifiche di lunghezza minima, ossia composte da *nb* bit;
- si effettua una sintesi a due livelli delle funzioni di stato prossimo e di uscita;
- si utilizzano bistabili di tipo *DT*;
- si cercano codifiche degli stati che minimizzino la logica combinatoria.

Con queste ipotesi, assegnare opportunamente a coppie di stati codifiche adiacenti (a distanza di Hamming unitaria) può permettere di avere mintermini a distanza di Hamming unitaria nella rappresentazione in somma di prodotti delle funzioni λ e δ. Questa condizione permette di ridurre il costo delle funzioni λ e δ poiché i loro mintermini possono essere raggruppati in prodotti di costo inferiore.

In base a questo criterio, considerando separatamente la sintesi delle funzioni di stato prossimo e di uscita si ottengono i seguenti criteri di scelta delle codifiche:

1) è opportuno che due stati s_i e s_j che, a parità di ingressi, hanno gli stessi stati prossimi abbiano codifiche adiacenti; si ottiene così una semplificazione della sintesi combinatoria della funzione δ;
2) è opportuno che due stati s_i e s_j che, a parità di ingressi, hanno le stesse uscite abbiano codifiche adiacenti; si ottiene così una semplificazione della sintesi combinatoria della della funzione λ;
3) è opportuno che due stati s_j e s_k tali che sia $s_j = \delta(s_i,i_\alpha)$ e $s_k = \delta(s_i,i_\beta)$, dove i_α e i_β sono codifiche d'ingresso a distanza di Hamming unitaria, abbiano codifiche adiacenti; si ottiene così una semplificazione della sintesi combinatoria della funzione δ.

Le regole 1 e 2 mirano a semplificare le dipendenze delle funzioni dalle variabili di stato, la 3 a semplificare le dipendenze dalle variabili d'ingresso primnarie.

Questi criteri sono estremamente generici e con buona probabilità possono portare a proposte di assegnamento non reciprocamente compatibili. È utile raggruppare questi criteri in un **grafo delle adiacenze** che possiede un nodo per ogni stato ed un arco per ogni coppia di stati che dovrebbero avere codifica adiacente. Soddisfare i criteri indicati significa dare assegnamenti adiacenti a ogni coppia di nodi collegati da un arco.

Riprendiamo la FSM di tabella 4.27 qui riportata per comodità.

	0	**1**
1	3 / 00	2 / 01
2	2/ 10	1 / 11
3	1 / 01	2 / 11

Il criterio 1, applicato alla colonna x=1, porta a chiedere l'adiacenza dei due stati 1 e 3 (ambedue hanno come stato prossimo 2); dalla colonna x=0 non vengono posti vincoli. Il criterio 2 porta a chiedere (relativamente all'uscita z1) l'adiacenza fra 1 e 3 e quella fra 2 e 3 e (per l'uscita z2) quella fra tutti e tre gli stati. Il criterio 3 porta a chiedere le adiacenze, rispettivamente, fra 2 e 3 e fra 2 e 1. A questi vincoli corrisponde il grafo delle adiacenze in figura 4.15.

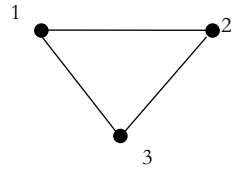


Figura 4.15. *Grafo delle adiacenze della FSM in tabella 4.27.*

Non esiste nessun assegnamento con due sole variabili che soddisfi tutti i vincoli (c'è un percorso triangolare chiuso, e non esistono percorsi di tale tipo su nessun *n-cubo*); occorre quindi "rinunciare" a qualche vincolo. Decidiamo di soddisfare il primo e il secondo vincolo ponendo 1=00, 2=11, 3=10; si ottiene la tabella delle transizioni 4.34.

	0	**1**
00	10 / 00	11 / 01
11	11/ 10	00 / 11
10	00/ 01	11 / 11

Tabella 4.34. *Tabella delle transizioni della FSM di tabella 4.27.*

Da questa tabella delle transizioni (considerando bistabili di tipo D) si ricava la seguente sintesi:

Y1=x'y1'+x'y2+xy2'

Y2=x'y2+xy2'

z1=y2+xy1

z2=x+y1y2'

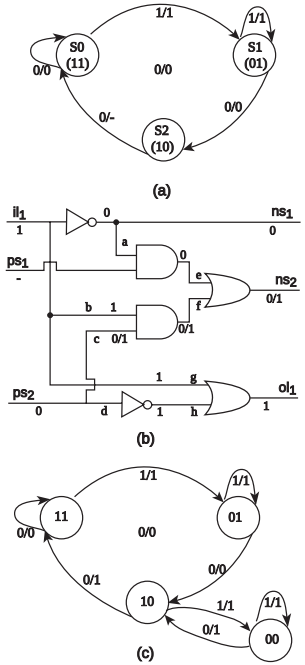
4.8 MINIMIZZAZIONE LOGICA

L'assegnamento di codifiche agli stati della FSM permette di esprimere in somma di prodotti le funzioni di stato prossimo e di uscita come già descritto nel paragrafo 4.5.2. Questa rappresentazione a due livelli deve essere considerata come il punto di partenza per effettuare una sintesi multilivello della logica combinatoria secondo gli obiettivi già descritti nel capitolo 2. Il fatto però che alcune linee di ingresso della logica combinatoria non siano effettivamente ingressi principali, ma provengano da elementi di memoria, fa sì che si possa no applicare algoritmi di minimizzazione particolari.

Infatti, nel caso in cui $|S| < 2^{nb}$, la fase di assegnamento delle codifiche agli stati identifica 2^{nb}- $|S|$ codifiche non utilizzate. Queste codifiche rappresentano valori che non potranno mai essere presenti negli elementi di memoria, poiché corrispondono a stati non appartenenti alla FSM, **non** è quindi possibile che, durante la normale evoluzione del circuito sequenziale, vengano caricate queste codifiche negli elementi di memoria. L'insieme delle codifiche di stato non utilizzate può essere anche visto come l'insieme degli stati non raggiungibili dallo stato di reset del circuito, identificati attraversando a livelli la FSM corrispondente al circuito sequenziale come già visto per la verifica di equivalenza di due FSM.

La corrispondenza tra gli stati non specificati e gli stati non raggiungibili è sicuramente vera nel caso in cui la FSM sintetizzata sia completamente specificata o sia parzialmente specificata soltanto per ciò che riguarda le uscite. Nel caso in cui invece la non specifica riguardi lo stato prossimo di alcune transizioni, potrebbe succedere che la fase di minimizzazione logica assegni a questi stati prossimi delle codifiche corrispondenti a stati non specificati, rendendo quindi raggiungibili stati

non specificati. Si consideri, per esempio, la seguente FSM parzialmente specificata in cui la transizione uscente dallo stato *S2* in corrispondenza dell'ingresso *1* non è specificata.



Utilizzando come assegnamento degli stati quello riportato in figura, si può ottenere il circuito mostrato in cui la fase di minimizzazione logica ha casualmente assegnato come stato prossimo di *10* (stato *S2*), in corrispondenza dell'ingresso *1*, la codifica di stato *00*. La codifica *00* corrisponde ad uno stato non specificato che risulta così raggiungibile dallo stato di reset *S0*. Di seguito è infatti mostrato lo STG estratto dal circuito sintetizzato che è composto da 4 stati invece dei 3 originali. In questo caso quindi l'insieme degli stati specificati è inferiore all'insieme degli stati raggiungibili.

Chiarito questo aspetto si può comunque affermare che:

- l'insieme delle codifiche di stato non utilizzate di una FSM, o
- l'insieme degli stati non raggiungibili dallo stato di reset di un circuito sintetizzato da una FSM,

rappresentano un insieme di valori che non potranno mai essere presenti negli elementi di memoria durante il funzionamento corretto del circuito e se presenti

rappresentano stati non appartenenti alla specifica. Queste codifiche possono quindi essere utilizzate come condizioni di indifferenza (*don't cares*) durante l'ottimizzazione della logica combinatoria. Infatti, a partire da questi stati presenti, per ogni ingresso, gli stati prossimi e le uscite sono indifferenti. Il problema maggiore consiste nell'identificazione degli stati non raggiungibili. Se si possiede la descrizione della FSM il problema è facilmente risolvibile poiché le codifiche di stato non utilizzate sono quelle non assegnate agli stati. Se invece si vogliono identificare gli stati raggiungibili di un circuito sintetizzato, il problema può essere brillantemente risolto utilizzando tecniche implicite come quelle utilizzate per la verifica di equivalenza di due FSM.

4.9 LA SINTESI SEQUENZIALE MEDIANTE SIS

Il progetto di macchine sequenziali, al di sopra delle modeste complessità (dell'ordine, al massimo, della decina di stati) che possono essere gestite in modo efficiente con tecniche manuali, viene oggi normalmente supportato da pacchetti CAD che implementano le metodologie descritte in questo capitolo. In particolare, in questo corso faremo riferimento a un programma sviluppato all'Università di California a Berkeley denominato SIS; questo si basa su una descrizione "testuale" di componenti (*model*) su cui si possono eseguire le varie operazioni di ottimizzazione. Questo tipo di rappresentazione può essere gerarchica e si basa sul formato *blif* (Berkeley Logic Interchange Format) già descritto nei capitoli precedenti nel caso di circuiti combinatori.

Per introdurre la progettazione di circuiti sequenziali mediante SIS, si parta da un esempio molto semplice: quello di un *semaforo con priorità*.

Si consideri il circuito di controllo di un semaforo posto all'incrocio tra una strada principale (direttrice Nord-Sud, NS) e una secondaria (direttrice Est-Ovest, EO). L'incrocio è dotato di sensori che rivelano la presenza di traffico; per semplicità, si consideri che i semafori commutino solo tra i colori verde e rosso. Il circuito ha due segnali di ingresso (TRAFFICONS e TRAFFICOEO di 1 bit) e due segnali di uscita (LUCENS e LUCEEO di 1 bit) con il seguente significato.

TRAFFICONS[1]: quando vale 1 segnala la presenza di traffico lungo la direttrice NS.

TRAFFICOEO[1]: quando vale 1 segnala la presenza di traffico lungo la direttrice EO.

LUCENS[1]: se posto a 1 fa accendere la luce verde nei semafori della direttrice NS, se posto a 0 fa accendere la luce rossa.

LUCEEO[1]: se posto a 1 fa accendere la luce verde nei semafori della direttrice EO, se posto a 0 fa accendere la luce rossa.

Il circuito di controllo deve garantire innanzitutto che le luci dei semafori delle direttrici NS e EO siano sempre accese in opposizione. Il circuito assegna priorità alla direttrice NS e commuta dal

verde al rosso questa direttrice solo se TRAFFICONS vale 0 e TRAFFICOEO vale 1; altrimenti mantiene il verde sulla direttrice NS. In assenza di traffico i semafori non modificano la configurazione raggiunta. Inoltre, non appena si presenta del traffico lungo la direttrice NS, il controllore pone a verde i semafori di questa direttrice e a rosso i semafori della direttrice EO indipendentemente dalla presenza di traffico sulla direttrice EO.

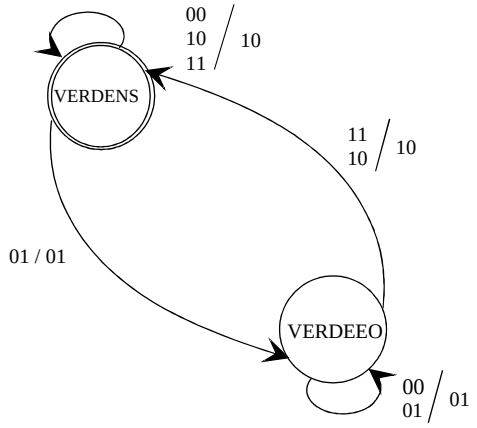


***Figura 4.16**. Diagramma degli stati del controllore del semaforo con priorità.*

In figura 4.16 è riportato il diagramma degli stati del semaforo con priorità relativo alla specifica informale data precedentemente. La tabella degli stati, corrispondente al diagramma degli stati di figura 4.12, è riportata in tabella 4.35.

	00	**01**	**10**	**11**
VERDENS	VERDENS / 10	VERDEEO / 01	VERDENS / 10	VERDENS / 10
VERDEEO	VERDEEO / 01	VERDEEO / 01	VERDENS / 10	VERDENS / 10

***Tabella 35.** Tabella degli stati del controllore del semaforo con priorità.*

4.9.1 Dalla FSM al circuito sequenziale

Il modello *.blif* del semaforo con priorità è il seguente:

```
.model SEMAFORO
.inputs TRAFFICONS TRAFFICOEO
.outputs LUCENS LUCEEO
.end
```

Se si crea il file di testo `semaforo.blif` con la descrizione del modello, è possibile utilizzare SIS per eseguire i comandi di base (in grassetto):

```
UC Berkeley, SIS 1.3
  (compiled 31-Dec-98 at 11:17 AM)
sis> read_blif semaforo.blif
```


Il modello del semaforo viene caricato in memoria.

```
sis> print_stats
SEMAFORO  pi= 2 po= 2 nodes= 2  latches= 0
lits(sop)= 0
```

Il modello in memoria si chiama `SEMAFORO`, a 2 bit di ingresso (`pi`), 2 bit di uscita (`po`), 0 elementi di memoria (`latches`) e la network associata è composta da 2 nodi (`nodes`) a cui non sono associate equazioni poiché il numero di letterali (`lits`), calcolato in somma di prodotti (`sop`), è zero.

```
sis> write_blif
.model SEMAFORO
.inputs TRAFFICONS TRAFFICOEO
.outputs LUCENS LUCEEO
.names LUCENS
.names LUCEEO
.end
```

La lettura del modello ha quindi fatto sì che SIS creasse una network in cui le uscite sono poste a zero come si vede stampando sullo schermo il modello corrente.

Inseriamo ora nel nostro modello la tabella delle transizioni tra le parole chiave `.start_kiss` e `.end_kiss`. Bisogna innanzi tutto ripetere quanti bit di ingresso e di uscita ha il nostro modello (`.i` e `.o`), quanti sono gli stati (`.s`), quante le transizioni (`.p`) e qual è lo stato di reset (`.r`). Ogni transizione viene descritta da una quadrupla ingresso/stato presente/stato prossimo/uscita. Si ottiene la seguente descrizione:

```
.model SEMAFORO
.inputs TRAFFICONS TRAFFICOEO
.outputs LUCENS LUCEEO
.start_kiss
.i 2
.o 2
.s 2
.r VERDENS
.p 5
00 VERDENS VERDENS 10
01 VERDENS VERDEEO 01
1- VERDENS VERDENS 10
0- VERDEEO VERDEEO 01
1- VERDEEO VERDENS 10
.end_kiss
.end
```

Rileggendo questo modello e stampandone le informazioni si ottiene ora:

```
sis> print_stats
SEMAFORO  pi= 2 po= 2 nodes= 2  latches= 0
lits(sop)= 0  #states(STG)= 2
```

si vede che sono stati riconosciuti 2 stati.

Per verificare la correttezza del modello descritto è anche possibile simularlo (a partire dal suo stato di reset) assegnando in successione dei valori agli ingressi con il comando `simulate`. Per esempio, a fronte della sequenza di ingresso `00 01 01`

10, la FSM evolve tra gli stati e genera le uscite nel seguente modo:

```
sis> simulate 0 0
Network simulation:
Outputs: 0 0
Next state:
STG simulation:
Outputs: 1 0
Next state: VERDENS ((null))

sis> simulate 0 1
Network simulation:
Outputs: 0 0
Next state:
STG simulation:
Outputs: 0 1
Next state: VERDEEO ((null))
```

```
sis> simulate 0 1
Network simulation:
Outputs: 0 0
Next state:
STG simulation:
Outputs: 0 1
Next state: VERDEEO ((null))

sis> simulate 1 0
Network simulation:
Outputs: 0 0
Next state:
STG simulation:
Outputs: 1 0
Next state: VERDENS ((null))
```

Si noti che la network associata al modello non rappresenta ancora la FSM infatti le sue uscite sono fissate a zero e gli stati non sono riportati. La simulazione dello STG è invece corretta, come è possibile verificare confrontando la figura 16.

È possibile generare automaticamente la network che realizza la FSM in modo che la simulazione della network e dello STG coincidano.

La sintesi non ottimizzata si basa sull'identificazione di una codifica binaria casuale per tutti gli stati di una FSM è sulla conseguente scrittura in somma di prodotti delle funzioni di stato prossimo e di uscita. È possibile utilizzare SIS per eseguire queste operazioni in modo da ottenere una prima realizzaione di una FSM da utilizzare come riferimento della sintesi ottimizzata che vedremo nel seguito. Si consideri ancora il controllore del semaforo con priorità temporizzato il cui STG è riportato in figura 4.18. La sua descrizione in formato *.blif* è stata riportata precedentemente. Assegnamo ai quattro stati della FSM la seguente codifica casuale a lunghezza minima, quindi su 2 bit:

	codifica
VERDENS	00
INIZIO CONTO	01
VERDEEO	10
INIZIO CONTO2	11

In formato *.blif* queste codifiche sono descritte mediante la parola chiave `.code` seguita dal nome dello stato e dalla codifica assegnata. I codici di stato vanno inseriti alla fine della descrizione della macchina a stati (`.end_kiss`) e prima della fine del modello (`.end`). Per questo esempio si ottiene:

```
.code VERDENS 00
.code INIZIO_CONTO 01
.code VERDEEO 10
.code INIZIO_CONTO2 11
```


Caricato con SIS questa nuovo versione del file `controllo.blif`, il comando `stg_to_network` costruisce automaticamente le funzioni di stato prossimo e di uscita. Viene costruita la network che realizza la FSM istanziando innanzi tutto due elementi di memoria per memorizzare le codifiche degli stati (il primo elemento di memoria corrisponde al bit di stato più significativo):

```
.latch [3] LatchOut_v3 0
.latch [4] LatchOut_v4 0
```

Questi elementi di memoria si resettano al valore 0 poché lo stato di reset (VERDENS) ha codifica 00. Le linee di stato prossimo sono state chiamate `[3]` e `[4]` mentre quelle di stato presente `LatchOut_v3` e `LatchOut_v4`. Inoltre, ogni linea di stato prossimo e ogni bit di uscita sono stati espressi in somma di prodotti nel seguente modo:

```
.names TRAFFICONS FINE LatchOut_v3 LatchOut_v4 [3]
--10 1
-011 1
0101 1

.names TRAFFICONS TRAFFICOEO FINE LatchOut_v3 LatchOut_v4 [4]
--011 1
1--10 1
0-001 1
01-00 1

.names TRAFFICONS FINE LatchOut_v3 LatchOut_v4 LUCENS
1-0- 1
--00 1
-111 1
0001 1

.names TRAFFICONS FINE LatchOut_v3 LatchOut_v4 LUCEEO
--10 1
-011 1
0101 1

.names TRAFFICONS TRAFFICOEO LatchOut_v3 LatchOut_v4 INIZIO
1-10 1
0100 1
```

È facile verificare la correttezza di questa operazione confrontando il grafo delle transizioni di figura 4.18 e le codifiche assegnate agli stati. Chiedendo le statistiche della network così generata (comando `print_stats`) si ottiene:

```
CONTROLLO  pi= 3 po= 3 nodes= 5  latches= 2
lits(sop)= 50 #states(STG)= 4
```

Il costo di questa rete è quindi di 50 letterali come si può verificare contando il numero di 0 e 1 dei 16 prodotti che compongono le 5 equazioni.

Una realizzazione di questo circuito, utilizzando porte AND e OR a 2 ingressi, richiede con un semplice calcolo 15 porte OR (16 prodotti - 1), 34 porte AND (50 letterali - 16 prodotti) più 5 porte NOT (3 ingressi + 2 elementi di memoria). Questo calcolo può essere fatto anche in maniera automatica e più raffinata eseguendo i seguenti comandi:

```
sis> rlib andornot.genlib
```

```
sis> map -s
sis> write_blif -n controllo.andornot.blif
```

Col primo comando si carica la libreria tecnoclogica che contiene solo le porte AND OR e NOT, col secondo si esegue il *mapping* tecnologico che riscrive le equazioni generiche in funzione delle porte della libreria ed infine col terzo si salva la network nel file `controllo.andornot.blif` indicando esplicitamente (opzione `-n`) i nomi delle porte utilizzate. L'ispezione di questo file permette di stabilire il numero delle porte utilizzate che sono leggermente inferiori a quelle stimate precedentemente, poiché l'algoritmo di *mapping* esegue anche qualche ottimizzazione logica. Infine, il comando `map` ha stimato la seguente area per il circuito creato:

```
total gate area:  79592.00
```

Tutte queste misure permettono di confrontare questa sintesi non ottimizzata con i risultati delle ottimizzazioni descritti nel seguito.

4.9.2 La minimizzazione degli stati

Un problema legato alla ricerca della FSM minima equivalente è quello di verificare l'equivalenza di due circuiti sequenziali. Dal punto di vista progettuale, questo operazione può essere utile nel caso si voglia verificare l'equivalenza di due circuiti progettati partendo da specifiche diverse nella forma o utilizzando programma di sintesi diversi. SIS fornisce a questo scopo il comando `verify_fsm` che verifica l'equivalenza di due circuiti sequenziali verificando l'equivalenza delle FSM corrispondentii. Il comando estrae dalle network dei due circuiti i due grafi delle transizioni. Siccome le FSM estratte da un circuito sono ovviamente completamente specificate (un circuito non può generare un'uscita non specificata), la verifica di equivalenza dei circuiti si basa sul concetto di indistinguibilità.

La verifica di equivalenza viene portata a termine attraversando a livelli (*breadth first*) le due FSM a partire dallo stato di reset di entrambe. Per ogni stato vengono applicati tutti i possibili simboli di ingresso identificando tutti gli stati prossimi e generando tutte le uscite. Se le uscite differiscono i due circuiti non sono equivalenti, se non differiscono la verifica viene spostata a verificare l'indistinguibilità degli stati raggiunti. Il processo termina quando non è possibile raggiungere stati che non siano già stati raggiunti, ossia quando l'attraversamento a livelli ha permesso di raggiungere tutti gli stati delle due FSM.

Per semplificare la verifica di uguaglianza delle uscite, a parità di stato e ingresso, in realtà si fondono i due circuiti in un unito circuito secondo lo schema riportato in figura 4.17. Il circuito così ottenuto ha gli stessi ingressi dei due circuiti originali, ma ha una unica uscita corrispondente allo XOR di tutte le uscite corrispondenti dei due circuiti. Questa unica uscita vale 1 quando almeno due uscite corrispodenti hanno valore diverso, ossia quando i due circuiti non sono equivalenti.

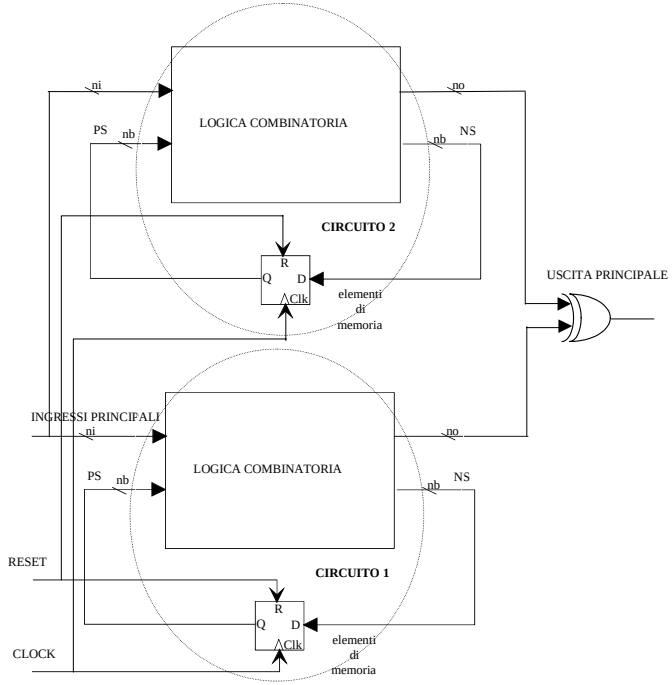


Figura 4.17. *Verifica di equivalenza di due circuiti sequenziali.*

La verifica di equivalenza si riduce quindi ad attraversare a livelli l'unica FSM estratta dal circuito composto e verificare che l'unica uscita valga sempre 0. Si noti soltanto che questa verifica fatta utilizzando una rappresentazione esplicita della FSM, tipo tabella delle transizioni, permetterebbe di verificare l'equivalenza di circuiti con pochi ingressi e pochissini elementi di memoria poichè per ogni gruppo di stati raggiunti durante l'attraversamento, devono essere provate tutte le configurazioni di ingresso e l'attraversamento termina quando tutti gli stati sono stati raggiunti. Utilizzando invece rappresentazioni più efficienti di una FSM, come quelle basate sui diagrammi di decisione binaria (*binary decision diagram*, *BDD*) è possibile verificare l'equivalenza di circuiti con decine di ingressi ed elementi di memoria. Il comando `verify_fsm` di SIS si basa sull'utilizzo di BDD.

Come esempio si utilizzi il comando `verify_fsm` per confrontare il circuito sequenziale del controllore del semaforo con priorità temporizzato prima e dopo le fasi di *mapping* tecnologico descritte in precedenza. Si hanno quindi due file: `controllo.blif` e `controllo.andornot.blif`. La sequenza dei comandi per verificare se i due circuiti sono equivalenti è la seguente:

```
sis> read_blif controllo.blif
sis> verify_fsm controllo.andornot.blif
```

Il risultato prodotto da SIS è il seguente:

```
The finite state machines have the same sequential behavior.
```

Il problema della identificazione della FSM minima equivalente viene affrontato in SIS richiamando un programma ausiliario: `stamina`. Stamina applica l'algoritmo di Paull-Unger in maniera molto efficiente usando matrici sparse per descrivere la tabella delle implicazioni e rappresentando le classi degli stati equivalenti, o compatibili, mediante tabelle hash. In questo modo i risultati prodotti da stamina sono molto vicini all'ottimo.

Per utilizzare `stamina` su un esempio più interessante di quello riportato in figura 4.16, si consideri lo STG del semaforo con priorità temporizzato, il cui STG è riportato in figura 4.18, e che verrà utilizzato e descritto nel capitolo 7.

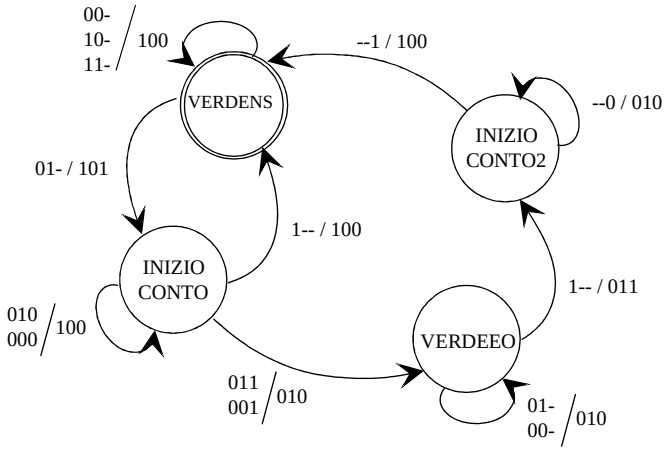


Figura 4.18. *Diagramma degli stati del controllore del semaforo con priorità temporizzato.*

Stanina lavora sullo STG associato alla network corrente riducendone, se possibile, il numero degli stati. Applicando `stamina` al controllore del semaforo con priorità temporizzato, il cui STG è riportato in figura 4.18, si ottengono i seguenti risultati:

```
sis> read_blif controllo.blif
```

```
sis> state_minimize stamina
Running stamina, written by June Rho, University of Colorado at
Boulder
Number of states in original machine : 4
Number of states in minimized machine : 4
```

```
sis> write_kiss
.i 3
```



```
.o 3
.p 10
.s 4
.r VERDENS
00- VERDENS VERDENS 100
1-- VERDENS VERDENS 100
01- VERDENS INIZIO_CONTO 101
0-0 INIZIO_CONTO INIZIO_CONTO 100
1-- INIZIO_CONTO VERDENS 100
0-1 INIZIO_CONTO VERDEEO 010
0-- VERDEEO VERDEEO 010
1-- VERDEEO INIZIO_CONTO2 011
--0 INIZIO_CONTO2 INIZIO_CONTO2 010
--1 INIZIO_CONTO2 VERDENS 100
```

Come si può vedere la tabella delle transizioni (in formato `kiss`) non è stata modificata poiché questa FSM era già in forma minima.

Si esegua invece ora `stamina` sulla FSM parzialmente specificata descritta in tabella 4.28. Questa FSM viene descritta come le altre in formato `.blif`.

```
.model fsm_nonspec
.inputs A
.outputs O
.start_kiss
.i 1
.o 1
.s 5
.p 10
.r SA
0 SA SE 0
1 SA SA 0
0 SB SD 0
1 SB SB 0
0 SC SE -
1 SC SC 0
0 SD SA 1
1 SD SA 1
0 SE SA -
1 SE SB -
.end_kiss
.end
```

La sua minimizzazione con stamina produce il seguente risultato:

```
sis> read_blif fsm_nonspec.blif
sis> state_minimize stamina
Running stamina, written by June Rho, University of Colorado at
Boulder
Number of states in original machine : 5
Number of states in minimized machine : 2
```

La FSM di dimensione minima identificata è quindi composta da due stati e la tabella delle transizioni è stata modificata nel seguente modo:

```
sis> write_kiss
.i 1
.o 1
.p 4
.s 2
.r S0
0 S0 S1 0
```

```
1 S0 S0 0
0 S1 S0 1
1 S1 S0 1
```

Il risultato è conforme con quanto calcolato manualmente.

4.9.3 Assegnamento degli stati

La funzione di assegnamento degli stati in SIS viene fornita da un programma esterno (`jedi`) che analizza la FSM associat alla network corrente. Il programma identifica un assegnamento degli stati utilizzando i criteri prima descritti, inoltre, esegue per un certo numero di assegnamenti di stato la generazione della rappresentazione a due livelli delle funzioni di stato prossimo e di uscita ed sceglie quella a costo (numero di letterali) inferiore. Applicando `jedi` all'esempio del semaforo (file `controllo.blif`) si ottengono i seguenti risultati:

```
sis> read_blif controllo.blif
```

```
sis> state_assign jedi
Running jedi, written by Bill Lin, UC Berkeley
```

```
sis> print_stats
CONTROLLO  pi= 3 po= 3 nodes= 5  latches= 2
lits(sop)= 33 #states(STG)= 4
```

La realizzazione delle funzioni di stato prossimo e di uscita costa quindi **33** letterali utilizzando l'assegnamento degli stati identificato da `jedi`. Questo assegnamento è:

```
sis> write_blif
…
.code VERDENS 00
.code INIZIO_CONTO 01
.code VERDEEO 10
.code INIZIO_CONTO2 11
…
```

Si noti che questi assegnamenti sono compatibili con il grafo delle adiacenze identificato manualmente. Utilizzando quei codici si otterrebbe un costo soltanto lievemente superiore (**35** letterali). Al contrario il costo della rete costruita utilizzando l'assegnamento casuale proposto nel paragrafo 4.7 è invece sensibilmente superiore (**50** letterali). Questo per dimostrare ulteriormente quanto l'assegnamento degli stati influenzi la complessità della rete combinatoria.

4.9.4 Minimizzazione Logica

Per il momento si consideri di voler minimizzare l'area e si utilizzi la sequenza di passi di minimizzazione che in SIS viene descritta dallo `script.rugged`. Applichiamo lo script all'esempio del semaforo di cui sono stati appena identificate le codifiche di stato. Si ottiene:

```
sis> print_stats
CONTROLLO  pi= 3 po= 3 nodes= 5  latches= 2
lits(sop)= 33 #states(STG)= 4
```

```
sis> source script.rugged
```



```
sis> print_stats
CONTROLLO  pi= 3 po= 3 nodes= 4  latches= 2
lits(sop)= 17 #states(STG)= 4
```

Come si vede il punto di partenza è una rete composta da 5 nodi del costo di 33 letterali, che a fronte della minimizzazione logica porta da avere una rete del costo di 17 letterali (con una riduzione del 48.5%. Il numero di nodi è sceso a 4 poiché gli algoritmi di fattorizzazione hanno verificato che il primo bit di stato è equivalente all'uscita LUCENS come si vede dalla parziale stampa della network:

```
.model CONTROLLO
.inputs TRAFFICONS TRAFFICOEO FINE
.outputs LUCENS LUCEEO INIZIO
.latch LUCENS LatchOut_v3 1
.latch [56] LatchOut_v4 0
…
.names LUCENS LUCEEO
0 1
.names LatchOut_v3 [56] LatchOut_v4 INIZIO
-10 1
001 1
.names TRAFFICONS FINE LatchOut_v3 LatchOut_v4 LUCENS
1-1- 1
-01- 1
-1-0 1
.names TRAFFICONS TRAFFICOEO LatchOut_v3 LatchOut_v4 [56]
0--1 1
011- 1
.end
```

Inoltre, come era prevedibile dalle specifiche, il segnale *LUCEEO* (luce verde in direzione est-ovest) è negato rispetto al segnale *LUCENS* (luce verde in direzione nord-sud): questo impedisce che ci siano condizioni in cui entrambi i semafori hanno luce verde.

All'interno di SIS non esiste un meccanismo automatico per la scrittura delle condizioni di indifferenza corrispondenti alle codifiche di stato non utilizzate. L'operazione dovrebbe essere svolta manualmente dopo l'utilizzo del programma `jedi` che fornisce l'elenco degli stati (parola chiave `.codes`) assegnati. Esiste invece il comando `extract_sequential_dc` che, utilizzando diagrammi di decisone binaria, attraversa la FSM estratta da un circuito e identifica se presenti, gli stati non raggiungibili. Questo comando costruisce automaticamente anche le condizioni di indifferenza. Per esempio si riconsideri nuovamente la FSM non completamente specificata a 5 stati descritta in tabella 4.28. Utilizzando `jedi` per l'identificazione delle codifiche di stato e lo `script.rugged` per la minimizzazione della logica combinatoria, si ottiene un circuito con le seguenti caratteristiche:

```
sis> print_stats
fsm_nonspec  pi= 1 po= 1 nodes= 5  latches= 3
lits(sop)= 13 #states(STG)= 5
```

L'identificazione degli stati non raggiungibili produce il seguente risultato:

```
sis> extract_seq_dc
number of latches = 3 depth = 4 state visited = 5
sis> write_blif
…
.codes SA 010
.codes SE 110
.codes SB 011
.codes SD 000
.codes SC 111
…
.exdc
.inputs LatchOut_v3 LatchOut_v2 LatchOut_v1
.outputs [49] [50] [51] O
.names LatchOut_v3 LatchOut_v2 LatchOut_v1 dc[71]
-01 1
100 1
.names dc[71] O
1 1
.names dc[71] [49]
1 1
…
```

Si vede che le tre codifiche di stato non utilizzate (*-01 100*) corrispondono agli stati non raggiungibili. È quindi possibile esprimere come condizioni di indifferenza (descritte dopo la parola chiave `.exdc`) il fatto che quando le tre linee di stato presente (*LatchOut_v3 LatchOut_v2 LatchOut_v1*) assumono i valori *-01* o *100*, sia l'uscita *O* del circuito, sia le tre linee di stato prossimo (*[49] [50] [51]*) sono *don't care*. Questo viene espresso da SIS generando la linea intermedia chiamata *dc[71]*, che rappresenta il valore *don't care*, e assegnandola all'uscita e alle linee di stato prossimo.

A fronte di queste condizioni di indifferenza, l'ulteriore esecuzione dell'algoritmo di minimizzazione `script.rugged` produce il seguente risultato:

```
sis> source script.rugged
sis> print_stats
fsm_nonspec  pi= 1 po= 1 nodes= 5  latches= 3
lits(sop)= 10 #states(STG)= 5
```

Il numero di letterali è sceso da 13 a 10 confermando che la presenza di elementi di memoria può imporre dei vincoli ai valori delle linee di stato presente che permettono di identificare ulteriori di condizioni di indifferenza per semplificare maggiormente la logica combinatoria che realizza le funzioni di stato prossimo e di uscita.