APPUNTI DI METODOLOGIE DI PROGETTO

Mariagiovanna Sami

Fabrizio Ferrandi

Capitolo 1

FORMALISMI ADOTTATI

INSIEMI E RELAZIONI

Un *insieme* è una collezione di oggetti detti *elementi* o *membri* dell'insieme. La *cardinalità* $|S|$ di un insieme S corrisponde al numero degli elementi di S. In questa tesi si considerano solo insiemi finiti cioè di cardinalità finita, tra questi, due casi particolari sono l'insieme *vuoto* $\varnothing$, che non contiene nessun elemento, e l'insieme *universo* U, da cui vengono presi tutti gli elementi degli insiemi considerati. In questa tesi si considera elementare il significato dei seguenti operatori: *inclusione* ($\subseteq$), *inclusione stretta* ($\subset$), *complemento* ($\neg$), *intersezione* ($\cap$), *unione* ($\cup$), *differenza* ($-$), *prodotto cartesiano* ($\times$). Maggiori dettagli possono essere trovati in [27].

Si definisce *relazione binaria* R fra due insiemi A e B un sottoinsieme del prodotto cartesiano $A \times B$ e si dice che un elemento $a \in A$ ed un elemento $b \in B$ sono in relazione rispetto ad R, cioè $a\ R\ b$, se la coppia $(a, b) \in R \subseteq A \times B$.

FUNZIONE

Una *funzione* F da un insieme X ad un insieme Y, $F : X \rightarrow Y$, è una regola di corrispondenza fra i due insiemi X (*dominio*) e Y (*codominio*), tale che ad ogni elemento del dominio viene associato un unico elemento del codominio. Poiché la notazione $y = F(x)$ è equivalente a $(x, y) \in F$ si può dire che la funzione è una relazione in cui ogni elemento dell'insieme X appare una sola volta nelle coppie appartenenti alla relazione F.

Dato il sottoinsieme Z di X, si definisce *immagine* di Z rispetto alla funzione F l'insieme $IMG(F, Z)$ specificato nel seguente modo:

$$IMG(F, Z) = \{y \in Y \mid \exists x \in Z, y = F(x)\}$$

In maniera simmetrica, si può definire la *pre-immagine* di Z rispetto ad F:

$$PRE(F, Z) = \{x \in X \mid \exists y \in Z, y = F(x)\}$$

In generale l'uscita y di una funzione può essere scomposta in più uscite $y_1, ..., y_m$ in modo tale che $y_1 \times y_2 \times ... \times y_m = y$. La *proiezione* di una funzione F rispetto ad un'uscita y_i, indicata con $F_{|y_i}$, è uguale alla funzione derivata da F considerando solo l'uscita y_i.

Una funzione F è *suriettiva* quando esiste una corrispondenza tale che ogni elemento del codominio risulta essere immagine di almeno un elemento del dominio. Una funzione F è *iniettiva* quando esiste una corrispondenza tale che non esistono due elementi del dominio che hanno la stessa immagine. Una funzione è *bijettiva* quando è sia iniettiva che suriettiva.

FUNZIONE LOGICA

Le funzioni logiche, dette anche booleane, sono particolari funzioni basate su di una particolare algebra: l'algebra booleana. Le operazioni base dell'algebra booleana sono:

- l'operazione di negazione logica;

- l'operazione di congiunzione logica;
- le operazioni di disgiunzione logica inclusiva ed esclusiva.

Una proposizione è una asserzione che gode della proprietà di esprimere un giudizio di verità o di falsità. Le proposizioni vengono generalmente indicate con dei simboli che possono essere, ad esempio, le lettere dell'alfabeto. La risultante dell'applicazione di una operazione logica ad una o più proposizioni è anch'essa una proposizione. Una proposizione a cui venga applicata una operazione logica si dice *argomento* dell'operazione logica. La combinazione di una o più operazioni logiche è detta *funzione logica*.

Sia $B=\{0,1\}$ l'insieme booleano, siano x_1, x_2, ..., x_n n variabili booleane; si definisce funzione logica ad n variabili la funzione:

$$f(x_1,x_2,..,x_n) : B^n \rightarrow B$$

Il supporto di una funzione logica f è formato dall'insieme delle variabili logiche da cui f dipende. In una funzione logica, una variabile o la sua negazione si dice *letterale*. La congiunzione o prodotto di più letterali si definisce come *cubo*. Espandendo una generica funzione logica f utilizzando la decomposizione di Shannon (detta anche di Boole) si ottiene che la funzione f può essere espressa come somma di cubi detti *mintermini*.

L'*on-set* di una funzione logica f è costituito dall'insieme dei mintermini che rendono la funzione logica uguale ad 1, mentre l'*off-set* coincide con l'insieme dei mintermini che fanno assumere alla funzione il valore 0. Il *dc-set* di una funzione logica coincide con l'insieme dei mintermini per cui la funzione non risulta specificata.

Un vettore di funzioni logiche può essere utilizzato per rappresentare una funzione F che descrive una corrispondenza fra un generico insieme $X \subseteq B^n$ ed un insieme $Y \subseteq B^m$:

$$F(x_1,x_2,..,x_n) : B^n \rightarrow B^m$$

Cofattori, quantificazione e composizione

Data una funzione logica, $f(x_1,x_2,..,x_n)$ i *cofattori positivo e negativo* di f rispetto alla variabile x_i sono definiti rispettivamente da

$$f_{x_i} = f(x_1,x_2,..,x_{i-1},1,x_{i+1},..,x_n)$$

e da

$$f_{\neg x_i} = f(x_1,x_2,..,x_{i-1},0,x_{i+1},..,x_n)$$

Oltre alla definizione di cofattore si riporta anche la definizione di *cofattore generalizzato* ($\downarrow$ – *constraint*) di una funzione f rispetto ad un insieme A, come viene definita in [71] e in [34]:

$$\text{if } A(x_1,x_2,..,x_n) = 1 \quad \Rightarrow \quad f(x_1,x_2,..,x_n) \downarrow A = f(x_1,x_2,..,x_n)$$
$$\text{if } A(x_1,x_2,..,x_n) = 0 \quad \Rightarrow \quad f(x_1,x_2,..,x_n) \downarrow A = f(x_0)$$

dove $x_0 = \min_y d(x_1,x_2,..,x_n,y_1,y_2,..,y_n)$, $A(y_1,y_2,..,y_n) = 1$ e

$$d(x_1,x_2,..,x_n,y_1,y_2,..,y_n) = \sum_{1 \le i \le n} | xi - yi | \, 2^{n-i}$$

Quando l'insieme A rappresenta un sottoinsieme del dominio della funzione f l'operatore di cofattore generalizzato costruisce la funzione logica $f \downarrow A$ che ha la stessa immagine di f quando si considerano solo gli elementi dell'insieme A.

Data una funzione booleana $f(x_1,x_2,..,x_n)$ definita sullo spazio booleano B^n, il *quantificatore esistenziale* (o astrazione) di f rispetto ad x_i è definito come:

$$\exists_{x_i} f = f_{x_i} + f_{\neg x_i}$$

Analogamente il *quantificatore universale* è definito come:

$$\forall_{x_i} f = f_{x_i} \cdot f_{\neg x_i}$$

Infine, date due funzioni $g(x)$ e $f(y)$ dove $x = x_1,x_2,..,x_n$ e $y = y_1,y_2,..,y_m$, si ha che la *composizione* di due funzioni è descritta da $[g(x) \rightarrow y]\, f(y)$, rappresenta la funzione logica ottenuta sostituendo la funzione $g(x)$ alla variabile booleana y della funzione $f(y)$.

Rappresentazione di insiemi e di relazioni: funzione caratteristica

Sia B^n uno spazio booleano ad n dimensioni, un insieme di mintermini $S \subseteq B^n$ può essere rappresentato dalla *funzione caratteristica* di S, $\chi_S(x)$, per cui vale la seguente proprietà:

$$s \in S \Leftrightarrow \chi_S(s) = 1$$

per tutti gli $s \in B^n$.

In altri termini, un mintermine di B^n che rende χ_S uguale a 1 (cioè, un mintermine dell'on-set della funzione caratteristica) è anche un elemento dell'insieme S. La funzione caratteristica dell'insieme B^n (*universo*) corrisponde alla costante 1 mentre la funzione caratteristica dell'insieme vuoto coincide con 0. Se il supporto delle funzioni caratteristiche dei due insiemi S_1 e S_2 risulta essere disgiunto, la funzione caratteristica del prodotto cartesiano $S_1 \times S_2$ dei due insiemi S_1 e S_2 vale:

$$\chi_{S_1 \times S_2}(x, y) = \chi_{S_1}(x) \cdot \chi_{S_2}(y)$$

e presenta un insieme di supporto che è pari all'unione dei rispettivi supporti.

Sia $\Xi \subseteq A \times A$ una relazione fra gli elementi dell'insieme A, e siano x e y due elementi generici dell'insieme A; la funzione caratteristica $\chi_\Xi(x,y) = 1$, se e solo se l'elemento x è in relazione Ξ con l'elemento y, cioè:

$$x \,\Xi\, y \Leftrightarrow \chi_\Xi(x,y) = 1$$

per tutti gli $(x, y) \in \Xi$.

In questa tesi i termini insieme, relazione e funzione caratteristica saranno utilizzati in maniera intercambiabile in quanto verranno rappresentati utilizzando la stessa notazione.

Immagine e Pre-Immagine rispetto ad una relazione

Sia Q un sottoinsieme di A; l'*immagine* di Q rispetto alla relazione Ξ è uguale all'insieme degli elementi di A che sono in relazione con gli elementi di A contenuti in Q. In maniera formale si ha che:

$$IMG(\Xi, Q) = \exists_x (\Xi(x,y) \cdot Q(x))$$

In altre parole la formula sopra riportata indica che l'insieme degli elementi di A che sono in relazione Ξ con gli elementi di $Q \subseteq A$ si ottiene prima selezionando da Ξ tutte le coppie (x, y) tali che $x \in Q$, e poi eliminando da tali coppie gli x.

In maniera simile, si può definire l'operatore di *pre-immagine* di Q rispetto a Ξ come:

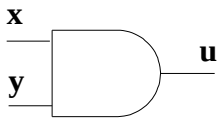
$$PRE\ (\Xi,Q) = \exists_y\ (\Xi(x,y) \cdot Q(y))$$

RAPPRESENTAZIONE DELLE FUNZIONI LOGICHE

Esistono diverse modalità di rappresentazione delle funzioni logiche; in questa tesi verranno considerate principalmente due forme: la *forma tabellare* e i *diagrammi di decisione binaria* (*Binary Decision Diagram – BDD* [94], [81]).

Forma tabellare

Il metodo più semplice per descrivere una funzione logica consiste nel definire una tabella, nota col nome di *tabella della verità*. Detto n il numero di operandi della funzione, la tabella della verità avrà $(n+1)$ colonne e (2^n) righe. Ciascuna riga riporta una delle (2^n) possibili combinazioni di valori degli operandi ed in corrispondenza della $(n+1)_{esima}$ colonna viene riportato il valore assunto dalla funzione quando si considera quella specifica combinazione di valori degli operandi.



(a)

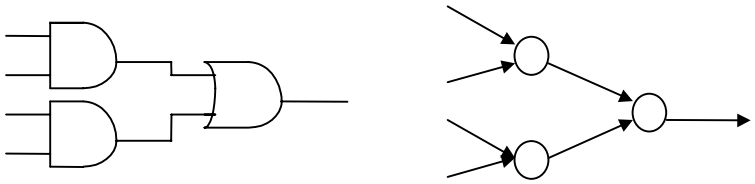
x	y	u
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Figura 1{Fig_tabella_ex}. Descrizione della funzione della porta logica *and* (a) attraverso la sua tabella della verità (b).

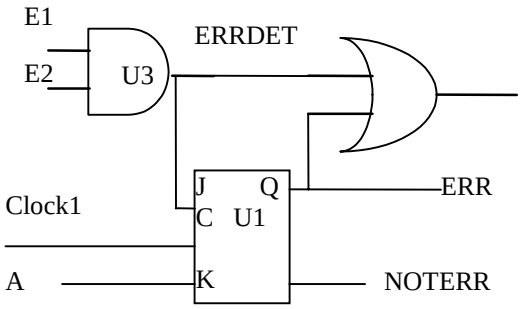
I modelli strutturali

Normalmente, con *modello strutturale* di un sistema digitale si indica una rappresentazione del suo schema logico. La rappresentazione esterna di un tale modello può avere forma grafica (lo schema logico) o testuale (nel caso si usi VHDL, sarà un programma in VHDL strutturale). Questo secondo tipo di rappresentazione farà riferimento a primitive e a *librerie di componenti*; un modello strutturale può includere informazioni temporali, in particolare per quanto riguarda i ritardi. In alcuni casi, del circuito interessano particolari proprietà legate alla *struttura* piuttosto che alle funzionalità dei componenti; in tal caso, il modello strutturale può essere ricondotto a un grafo. Nel caso di circuiti in cui i componenti *elaborino* informazione e le linee di segnale siano unidirezionali, si realizza un modello molto semplice costituito da un *grafo orientato* in cui componenti e linee di segnale ("nets") vengono rappresentati, rispettivamente, mediante nodi e lati orientati. Ad esempio, il seguente circuito a due livelli può essere schematizzato mediante un grafo i cui nodi rappresentano le porte logiche:

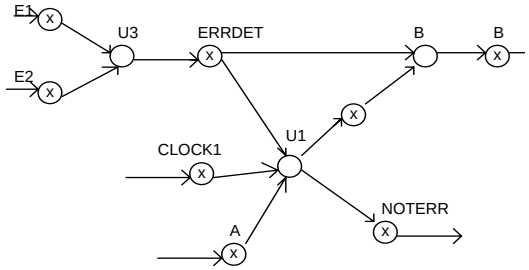


Figura

Nel circuito dato, ogni segnale si propaga da una sorgente a una sola destinazione: il circuito è cioè ***privo di fanout*** (fanout-free). In casi come questi, il grafo rappresentativo è sempre un albero. Se esiste fanout, per la rappresentazione si può ricorrere a ***grafi orientati bipartiti***, in cui *sia i componenti sia le linee vengono rappresentati mediante nodi*: in un tale grafo, ogni lato collega un nodo corrispondente a un componente a un nodo corrispondente a una linea. Mentre i nodi che rappresentano componenti possono avere più ingressi e più uscite, quelli corrispondenti alle linee possono avere un solo ingresso e più uscite. Esempio:



Figura

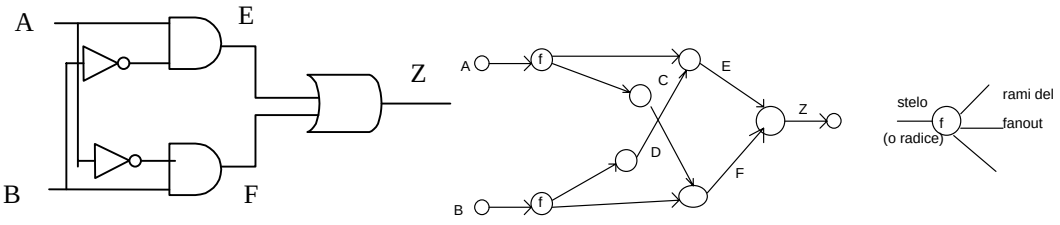


Figura

dove ○ è il simbolo di un nodo-componente, ⊗ il simbolo di un nodo-collegamento

Per circuiti in cui ogni componente abbia una sola uscita, ma in cui esistano punti di fanout, l'introduzione di nodi che rappresentano linee di collegamento fra due soli componenti (come nel grafo bipartito) risulta ampiamente ridondante: si giunge a grafi con un numero di nodi di gran lunga troppo alto, la cui elaborazione comporta un carico computazionale eccessivo e ingiustificato. Si preferisce ricorrere a grafi come quello strutturale semplice (in cui i collegamenti fisici sono rappresentati da lati), introducendo oltre ai nodi che rappresentano i

componenti nodi addizionali per rappresentare i soli punti di fanout (***nodi di fanout***, contrassegnati con una *f*) .



Figura

Lo schema precedente introduce una caratteristica che, come si vedrà, ha molta rilevanza per i problemi di collaudo, e cioè la presenza di ***fanout riconvergenti***, cioè più percorsi di segnale che da uno stesso segnale di origine riconvergono in un unico componente. Fanout riconvergenti la cui origine non sia su un ingresso primario vengono di solito introdotti nella fase di ottimizzazione di una sintesi a più livelli, o quando si realizza un circuito complesso componendo circuiti più semplici preesistenti.

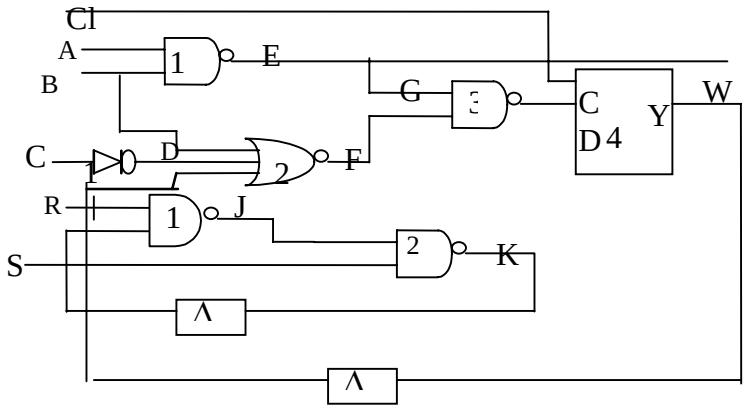
Su uno schema strutturale, si identificano alcuni parametri che risultano essenziali in fase di simulazione, simulazione di guasto, collaudo, etc. Un primo parametro è la ***parità d'inversione***, che si definisce relativamente ai percorsi presenti in circuiti costituiti solo da AND, OR, NOT, NAND e NOR come il numero (modulo 2) delle porte invertenti lungo il percorso. Prima di effettuare attività quali la simulazione logica, ai componenti di un circuito combinatorio vengono spesso associati i rispettivi *livelli logici* (o più semplicemente *livelli* - il circuito su cui sia stata fatta tale operazione viene detto *levelized*). Con ***livello*** di un elemento si indica la misura della sua *distanza dagli ingressi primari*: il livello degli ingressi primari è, per definizione, 0. Per un generico elemento *i*, con ingressi $k_1, k_2, \ldots k_p$, il livello *l(i)* è definito come

$$l(i) = 1 + \max_j l(k_j)$$

La procedura per determinare i livelli di un circuito combinatorio con tecnica "breadth-first" è:

1) assegnare *l(i)=0* a tutti gli ingressi primari;
2) per ogni elemento *i* il cui livello non è ancora specificato, ma tale che tutti gli elementi che ne forniscono gli ingressi abbiano un livello definito, calcolare *l(i)* secondo l'equazione data.

Il concetto di livello si può estendere a circuiti sequenziali, considerando che le linee di retroazione siano *pseudo-ingressi primari* e abbiano quindi livello 0. Si consideri il seguente esempio.



Figura

La determinazione dei livelli, compiuta secondo la regola prima indicata, porta ai seguenti risultati:

livello 1: E, D, J

livello 2: F,K

livello 3: G

livello 4: W

Invece di usare livelli espliciti, a volte si usano livelli impliciti ordinando gli elementi del circuito in modo che per qualsiasi coppia di elementi numerati *i1* e *i2* sia *i1*<*i2* se $l(i_1) \leq l(i_2)$

Binary Decision Diagram (BDD)

Le tabelle della verità, le mappe di Karnaugh ed in generale tutti i metodi di rappresentazione delle funzioni booleane basati sull'enumerazione dei mintermini, hanno la spiacevole proprietà di crescere in modo esponenziale rispetto al numero delle variabili coinvolte. Per mezzo dei BDD è possibile rimediare a questo pesante inconveniente e descrivere in modo semplice e conciso funzioni logiche molto complesse.

Si consideri la seguente funzione booleana:

$$f = A + (\neg B) \cdot C$$

e si assuma di voler definire una procedura per determinare il valore booleano di *f* dati i valori delle variabili *A*, *B* e *C*. Per far questo si può iniziare considerando il valore della variabile *A*. Se *A* = 1, allora *f* = 1, e si termina. Se *A* = 0 si deve considerare il valore di *B*; se *B* = 1 allora *f* = 0 e si termina, altrimenti si deve considerare anche *C* ed il suo valore sarà uguale al valore assunto da *f*. Il costrutto *se..allora..altrimenti*, a cui costantemente si è fatto ricorso per determinare il valore della funzione *f*, può essere efficacemente rappresentato mediante un grafo costruito con il seguente procedimento:

- si associa un nodo del grafo ad ogni variabile x_i da cui dipende la funzione *f*;

- da ogni nodo, etichettato con il nome di una variabile x_i, escono due archi orientati corrispondenti alle condizioni $x_i = 0$ e $x_i = 1$;*
- oltre ai nodi etichettati con il nome delle variabili, nel grafo esistono altri nodi con un'etichetta corrispondente ad una costante. Questi nodi non hanno archi uscenti, sono detti nodi *terminali* e la costante che li etichetta rappresenta il valore assunto dalla funzione considerata.

Verrà ora esaminata la descrizione di una funzione per mezzo del formalismo introdotto. Per semplicità, si consideri la funzione $f = A \cdot B$; se $A = 0$ allora $f = 0$; se $A = 1$ il valore di f dipende anche dal valore della variabile B; se $B = 0$ allora $f = 0$ altrimenti $f = 1$. Il risultato dell'applicazione del procedimento prima introdotto alla funzione $f = A \cdot B$ porta al grafo mostrato in figura 2.

Si supponga ora di disporre della espressione booleana della funzione f e di volere ricavare il grafo ad essa relativo. In questo caso si utilizza un procedimento di tipo *top-down* che permette di derivare il diagramma per mezzo di ripetute applicazioni della decomposizione di Shannon.

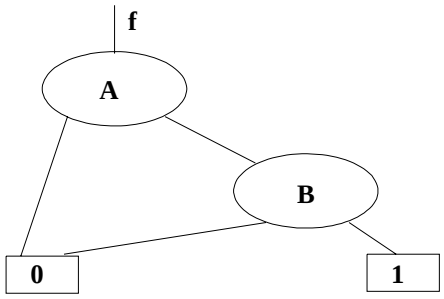


Figura 2{Fig_cap2_fig4}. Grafo corrispondente alla funzione $f = A \cdot B$.

In maniera formale, sia $f(x_1,x_2,..,x_n)$ una generica funzione booleana nelle variabili $x_1,x_2,..,x_n$, la decomposizione di Shannon della funzione logica f vale:

$$f(x_1,x_2,..,x_n) = x_1 \cdot f(1,x_2,..,x_n) + \neg x_1 \cdot f(0,x_2,..,x_n)$$

per tutte le variabili $x_1,x_2,..,x_n \in B^n$.

Facendo ricorso al concetto di *restrizione* o *cofattore* di f rispetto ad una variabile x_i, la decomposizione può essere riscritta come segue:

$$f(x_1,x_2,..,x_n) = x_1 \cdot f_{x_i} + \neg x_1 \cdot f_{\neg x_i}$$

Per esempio considerando la funzione:

$$g(a,b,c,d,e) = b \cdot (\neg a \cdot c + \neg c \cdot \neg e) + \neg e \cdot (\neg a \cdot b + \neg b \cdot d)$$

La decomposizione di Shannon rispetto alla variabile a produce:

$$g(a,b,c,d,e) = a \cdot g_a + \neg a \cdot g_{\neg a}$$

dove

* Nelle figure riportate nel seguito si assume *sempre* che l'arco uscente dal nodo verso sinistra corrisponda al valore 0 della variabile, l'arco uscente verso destra al valore 1.

$$g_a = b \cdot \neg c \cdot \neg e + \neg e \cdot \neg b \cdot d$$

$$g_{\neg a} = b \cdot (c + \neg c \cdot \neg e) + \neg e \cdot (b + \neg b \cdot d)$$

Iterando ulteriormente, considerando ad esempio la variabile *b*, si ottengono le funzioni che devono essere realizzate dai rami 0 e 1 dei nodi marcati con la variabile *b*. La figura 3 descrive la decomposizione dopo aver considerato la prima e la seconda variabile.

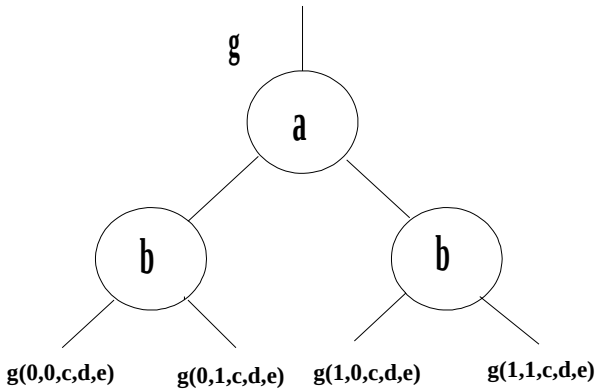


Figura 3{Fig_cap2_fig8}. Grafo della funzione *g* espanso rispetto alle variabili *a* e *b*.

Infine, durante questo procedimento di decomposizione alcune delle funzioni generate possono coincidere (nell'esempio riportato si può dimostrare che $g(0,1,c,d,e) = g(1,0,c,d,e)$), in questo caso gli archi corrispondenti possono essere diretti al medesimo nodo. Iterando i passi riportati (raggruppamento di sottofunzioni identiche ed espansione rispetto alle variabili non ancora considerate) si giunge infine al grafo completo della funzione *g*.

Chiaramente tutti i percorsi saranno lunghi al più *n* passi (nell'esempio riportato $n=5$) e termineranno in un nodo etichettato con uno 0 o con un 1.

Il grafo prodotto seguendo la decomposizione di Shannon produce un grafo noto come Binary Decision Diagram – BDD. Perché questa rappresentazione possa essere utilmente impiegata nella rappresentazione di funzione logiche è necessario che essa sia canonica (una rappresentazione che identifichi univocamente la funzione) e che i relativi operatori siano di complessità polinomiale. In letteratura esiste un caso particolare dei BDD, denominato *ROBDD* (*Reduced Ordered BDD*), che gode della proprietà di essere una rappresentazione canonica e di essere di semplice manipolazione.

Una semplice estensione dei ROBDD consente la rappresentazione delle normali funzioni (quelle che hanno codominio diverso da quello booleano). Infatti è sufficiente rappresentare una generica funzione attraverso un vettore di funzioni logiche tali che i sottografi comuni vengano condivisi. Implementativamente risulta tra l'altro semplice sfruttare le condivisioni fra diversi ROBDD in quanto è sufficiente che il gestore dei ROBDD durante la costruzione verifichi l'eventualità che un determinato sottografo sia già stato costruito. Nella realtà viene in pratica costruito un unico grafo che presenta tante radici quante sono le funzioni considerate e questo grafo presenta un unico ordinamento delle variabili in maniera da consentire la condivisione dei sottografi. In letteratura questa caratteristica viene evidenziata denominando i ROBBD con *multi-rooted DAG* ([46]) oppure con *Shared Binary Decision Diagram* (*SBDD*) ([90]).

Nei prossimi paragrafi si definiranno in maniera formale solo i *ROBDD* (nel seguito indicati semplicemente come BDD) riportandone proprietà, dimensioni e relative complessità computazionali.

Rappresentazione e proprietà dei BDD

Definizione Un BDD è un grafo diretto aciclico il cui insieme dei nodi V contiene solo due tipi di nodi, nodi *terminali* e nodi *non terminali*. Un nodo terminale v è un nodo avente come attributo un valore $Valore(v) \in \{0,1\}$. Un nodo non terminale v è un nodo avente come attributi un indice $Indice(v) \in \{1,2,..,n\}$ e due nodi figli $low(v)$ e $high(v)$. Nel caso in cui $v \in V$ sia non terminale e non lo sia anche $low(v)$ allora si ha che:

$$Indice(v) < Indice(low(v)).$$

Analogamente se $high(v)$ è un nodo non terminale si ha che:

$$Indice(v) < Indice(high(v)).$$

Inoltre, un qualsiasi grafo che ha per radice il nodo v identifica una funzione booleana f_v tale che:

- se v è un nodo terminale
  - se $Valore(v) = 1$ allora $f_v = 1$;
  - se $Valore(v) = 0$ allora $f_v = 0$;
- altrimenti se v non è un nodo terminale si verifica che

$$f_v(x_1,..,x_{i-1},x_i,x_{i+1},..,x_n) = x_i \cdot f_v(x_1,..,x_{i-1},1,x_{i+1},..,x_n) + \neg x_i \cdot f_v(x_1,..,x_{i-1},0,x_{i+1},..,x_n)$$

dove x_i è la variabile rappresentata dal nodo v mentre

$$f_v(x_1,..,x_{i-1},1,x_{i+1},..,x_n)$$

e

$$f_v(x_1,..,x_{i-1},0,x_{i+1},..,x_n)$$

sono rispettivamente i rami uscenti da v e corrispondenti a $x_i = 1$ ed $x_i = 0$.

Definizione Due grafi G e G' sono *isomorfi* se esiste una funzione biunivoca σ avente come dominio i nodi di G e come codominio i nodi di G', cosicché per ogni nodo v, se $\sigma(v) = v'$ allora o sia v che v' sono nodi terminali con valore $Valore(v) = Valore(v')$, o entrambi sono nodi non terminali tali che $Indice(v) = Indice(v')$, $\sigma(low(v)) = low(v')$ e $\sigma(high(v)) = high(v')$.

Definizione Dato un nodo v si definisce come *sottografo di radice v* il grafo costituito da v e da tutti i suoi discendenti.

Lemma Se G e G' sono isomorfi rispetto a σ allora il sottografo di radice v, per ogni v in G, è isomorfo con il sottografo di radice $\sigma(v)$.

Un grafo può essere ridotto senza modificare la funzione che esso rappresenta eliminando i vertici ridondanti e i sottografi duplicati nel grafo stesso.

Definizione Sia G il grafo associato alla funzione f, si dice che G è *ridotto* se valgono entrambe le seguenti condizioni:

- G non contiene nessun nodo v tale che $low(v) = high(v)$;

- G non contiene nodi distinti v e v' tali che i sottografi di radice v e v' siano rispettivamente isomorfi.

Lemma Per ogni nodo v in un grafo G ridotto, il sottografo di radice v è ancora un grafo ridotto.

Sussiste quindi il seguente teorema, che garantisce la canonicità dei ROBDD:

Teorema *Unicità del grafo ridotto* Una funzione booleana f è rappresentata da un unico grafo ridotto G. Inoltre, G possiede il minor numero di vertici rispetto a qualunque altro grafo che rappresenti f.

Per la dimostrazione del teorema si rimanda a [81].

Complessità computazionale delle operazioni implementate mediante BDD

Nel caso pessimo, il numero di nodi necessari per rappresentare una funzione logica mediante BDD cresce esponenzialmente con il numero delle variabili, e per la precisione cresce con $O(2^n/n)$ (per un'analisi dettagliata delle problematiche relative alla complessità di una rappresentazione basata su BDD si veda [33], [83], [40], [41]). Nella maggior parte dei casi pratici (sommatori, multiplexer, decodificatori, etc.) il numero dei nodi cresce *linearmente con il numero delle variabili*, perciò in genere una rappresentazione basata su BDD permette di ottenere un notevole risparmio di memoria.

Inoltre, la complessità degli operatori associati ai BDD è polinomiale nel numero dei nodi dei grafi su cui essi operano (come riassunto in tabella 1).

Operazione	Risultato	Complessità
Riduzione del grafo	G forma canonica	$O(\|G\| \cdot log\ (\|G\|))$
Applicazione operatore <op>	f_1<op>f_2	$O(\|G_1\| \cdot \|G_2\|)$
Calcolo cofattore	f_{x_i} oppure $f_{\neg x_i}$	$O(\|G\| \cdot log\ (\|G\|))$
Composizione di funzioni	$[f_2(x) \rightarrow y] f_1(y)$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Calcolo dell'insieme S_f	S_f	$O(n \cdot \|S_f\|)$
Calcolo di almeno 1 elemento in S_f	elementi in S_f	$O(n)$
Calcolo del numero di elementi in S_f	$\|S_f\|$	$O(\|G\|)$

Tabella 1 {Tab_complessita_bdd}Complessità computazionale delle principali operazioni eseguibili su BDD [81].

Nella tabella 1 il simbolo <*op*> indica una qualunque delle operazioni dell'algebra booleana (per esempio *and*, *or*, *not*, *xor*,...), mentre S_f rappresenta l'insieme dei mintermini dell'on-set di f.

Ordinamento

Il numero dei nodi di una rappresentazione basata su BDD, a parità di funzione booleana da rappresentare, dipende dall'ordinamento delle variabili coinvolte; si considerino, ad esempio, le due funzioni:

$$f_1(x_1,..,x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$$

e

$$f_2(x_1,..,x_6) = x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$$

che differiscono semplicemente per una permutazione delle variabili. La prima è rappresentata da un grafo di soli 6 nodi, la seconda da un grafo che presenta 14 nodi (rispettivamente in figura 4 (a) ed in figura 4 (b)).

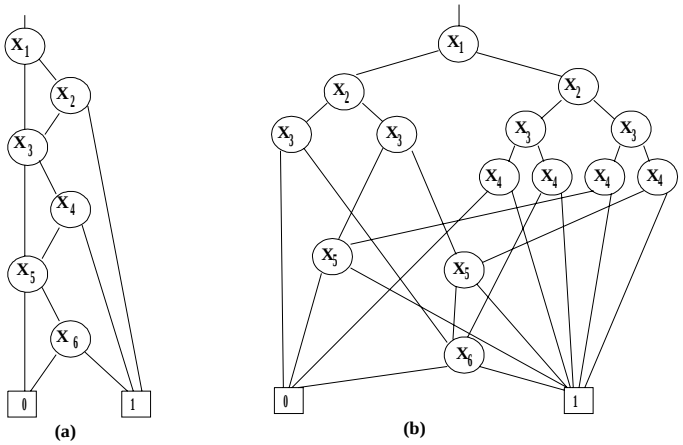


Figura 4{Fig_cap2_fig9}. Influenza dell'ordinamento delle variabili sulle dimensioni del BDD: grafo della funzione $f_1(x_1,..,x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ (a) e della funzione $f_2(x_1,..,x_6) = x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ (b).

Il problema dell'identificazione dell'ordine ottimo, che produce il BDD con il numero minimo di nodi, è un problema *NP-completo*. Si ricordi che un problema è *NP* se esiste un algoritmo che ne dà una soluzione in un tempo polinomiale (che cresce in maniera polinomiale al crescere della dimensione del problema) su di una macchina non deterministica; mentre si dice *NP-completo* se è *NP*, ma non è conosciuto nessun algoritmo che dà una soluzione in un tempo polinomiale su una macchina deterministica.

Nella pratica si è però osservato che si ottengono buoni risultati semplicemente applicando alcune regole euristiche basate sulla struttura del BDD stesso ([80], [91], [92]) o sulla struttura del circuito che rappresentano ([31], [60], [89]). In questa tesi non si considereranno algoritmi di riordino basati sulla struttura del circuito dato che le rappresentazioni dei circuiti adottate sono principalmente di tipo funzionale. Tra gli algoritmi di riordino funzionale presenti in letteratura si ricordano:

- **Sift**: è l'implementazione dell'algoritmo di *sift* [80]. L'algoritmo considera una variabile per volta e la sposta in tutte le possibili posizioni. La migliore posizione è quella che minimizza le dimensioni del grafo.
- **Sift simmetrico**: è come l'algoritmo *sift* con una aggiunta: viene verificata la simmetria delle variabili che, durante l'applicazione dell'algoritmo, divengono adiacenti. Se le variabili sono simmetriche (la dimensione del grafo non varia) vengono raggruppate insieme e l'algoritmo di *sift* viene iterato sul gruppo di variabili e non più sulla singola variabile.
- **Group sift**: è come l'algoritmo di *sift* ma iterato su gruppi di variabili (non necessariamente simmetriche, come nel *sift simmetrico*).

Le librerie di gestione dei BDD più sofisticate implementano algoritmi di ordinamento automatici, cioè è il gestore dei nodi allocati che decide autonomamente quando far partire l'algoritmo di riordino.

C a p i t o l o 2

DISPOSITIVO DIGITALE

IL PROBLEMA DEL PROGETTO: QUALITÀ, COSTI, ECONOMIA

Il settore dei dispositivi e sistemi digitali oggi vede:

- crescente diffusione, con applicazioni caratterizzate da volumi di produzione anche molto elevati (si pensi all'elettronica per auto, alle telecomunicazioni, alle applicazioni di tipo "consumer");
- crescente complessità dei dispositivi e dei sistemi: dal 1974 (anno dell'introduzione dell'INTEL 8008) a oggi, si verifica che ogni cinque anni la complessità dei dispositivi digitali integrati aumenta *di un ordine di grandezza* (all'inizio degli anni '80 I primi dispositivi "a grandissima integrazione" - VLSI - giungevano a 100.000 transistori su un chip; nel 1998 si trattano dispositivi integrati - che non siano memorie - di 100 *milioni* di transistori);
- crescente richiesta di qualità da parte degli utenti, sia "intermedi" (coloro che utilizzano il dispositivo/sistema come parte di un sistema più complesso) sia finali (anche utenti non professionali sono diventati sensibili al problema della qualità).

A fronte di questo, il mercato impone:

- tempi sempre più brevi per l'introduzione di una nuova "generazione" di prodotti - come conseguenza, i dispositivi esistenti vedono accelerare il cammino verso la maturità e la dismissione;
- vincoli sempre più pesanti sui costi - un esempio limite è rappresentato dal continuo calo dei costi dei Personal Computers, in proporzione alle crescenti prestazioni offerte.

Il secondo insieme di richieste è in buona misura contraddittorio rispetto al primo; per soddisfarlo, occorre *ridurre il tempo del ciclo di progettazione* (riducendo il "time to market", si soddisfa la pressione della crescente competitività), e al tempo stesso *diminuire tutti i costi su cui si possa intervenire* (che sono, oltre al costo di progetto, i costi per la verifica della qualità e per la successiva manutenzione: i costi dei processi produttivi, al contrario, non possono che aumentare in conseguenza sia della crescente complessità dei dispositivi sia degli stessi vincoli di qualità).

Il modello economico per il "time to market" può essere riassunto da un semplice grafico (fig. 1.1):

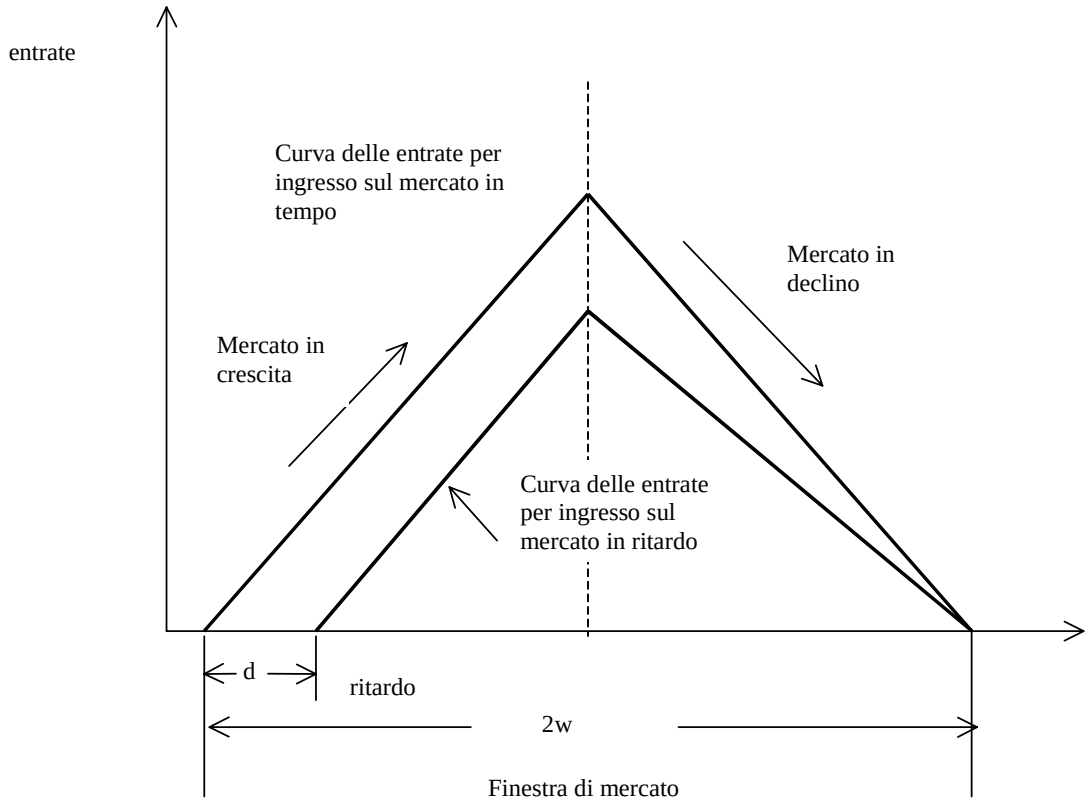


Fig. 1.1

Figura

La curva rappresenta le entrate dovute all'ingresso del prodotto sul mercato: la richiesta del prodotto cresce - dal momento della sua effettiva introduzione - fino a raggiungere un picco, poi diminuisce fino ad annullarsi. Se il prodotto compare sul mercato con un ritardo *d* rispetto al momento "ottimo" (tipicamente, quello dell'annuncio), si può poi prevedere che la richiesta del pubblico aumenti *con lo stesso tasso di crescita* che aveva nel caso ottimale - e che raggiunga il "picco" nello stesso istante - con la conseguenza che la richiesta di picco risulterà inevitabilmente inferiore (ad esempio, perché nel frattempo sono usciti prodotti concorrenti che propongono ai consumatori delle alternative). Peraltro, il calo della richiesta raggiungerà il momento di richiesta nulla nello stesso momento precedente; si avrà quindi una perdita economica approssimabile come segue:

$$\text{Entrate perse} = \text{Entrate totali previste} \times \left[\frac{d(3w-d)}{d^2}\right]$$

Si consideri un esempio che può essere considerato tipico, riferito a due diversi sistemi elettronici, ambedue con una vita di 18 mesi, il primo che porterebbe (se introdotto sul mercato all'istante giusto) entrate per 25 milioni di dollari, il secondo che (nelle stesse condizioni) porterebbe entrate di 50 milioni di dollari:

Entrate totali per il prodotto	25 M$	50 M$
vita del prodotto	18 mesi	18 mesi
entrate perse per ogni **giorno** di ritardo nell'ingresso sul mercato	**100 K$**	**200 K$**

Lo svantaggio di un'introduzione ritardata risulta ben chiaro!

Per potere soddisfare i requisiti e i vincoli citati in precedenza si rende indispensabile:

1) utilizzare strumenti di CAD capaci di rendere più veloce la fase di progetto, pur garantendo un livello di ottimizzazione comparabile a quello che raggiungerebbe "a mano" un progettista esperto (ad esempio, minimizzando i costi del prodotto, massimizzando la velocità di funzionamento, etc.);
2) portare la fase di progetto formale a livelli di astrazione sempre più elevati, introducendo tecniche di specifica rigorose e metodi formali per il passaggio dalla descrizione alla sintesi, così da poter utilizzare prima possibile strumenti di progettazione assistita da calcolatore; questo significa consentire al progettista di limitarsi a fornire la descrizione di funzionalità complesse e del comportamento del sistema a un alto livello di astrazione, senza costringerlo a fornire i dettagli di realizzazione;
3) introdurre tecniche di verifica automatica della correttezza del progetto, più soddisfacenti degli attuali metodi basati esclusivamente sulla simulazione, e comunque creare modelli di "errore di progetto" e corrispondenti tecniche di verifica in grado di esplorare in modo soddisfacente l'effettivo spazio di errore;
4) spingere al massimo la verifica del buon funzionamento del prodotto, prima di immetterlo sul mercato (operare cioè un *collaudo* il più possibile completo);
5) garantire la facilità di manutenzione del prodotto finale.

I punti 1,2,3 mirano a ridurre tempi e costo del progetto, garantendone al tempo stesso la correttezza (vista come corrispondenza alle specifiche iniziali): scopo finale di questi passi è raggiungere un progetto *"right first time"*, che non richieda di realizzare prototipi intermedi (procedura troppo costosa e di fatto inaccettabile con le moderne tecnologie elettroniche) né tanto meno di compiere successivi interventi di modifica e messa a punto. La verifica di correttezza contribuisce anche al raggiungimento di un'elevata *qualità* del prodotto finale, requisito che rende indispensabili i punti 4 ed 5.

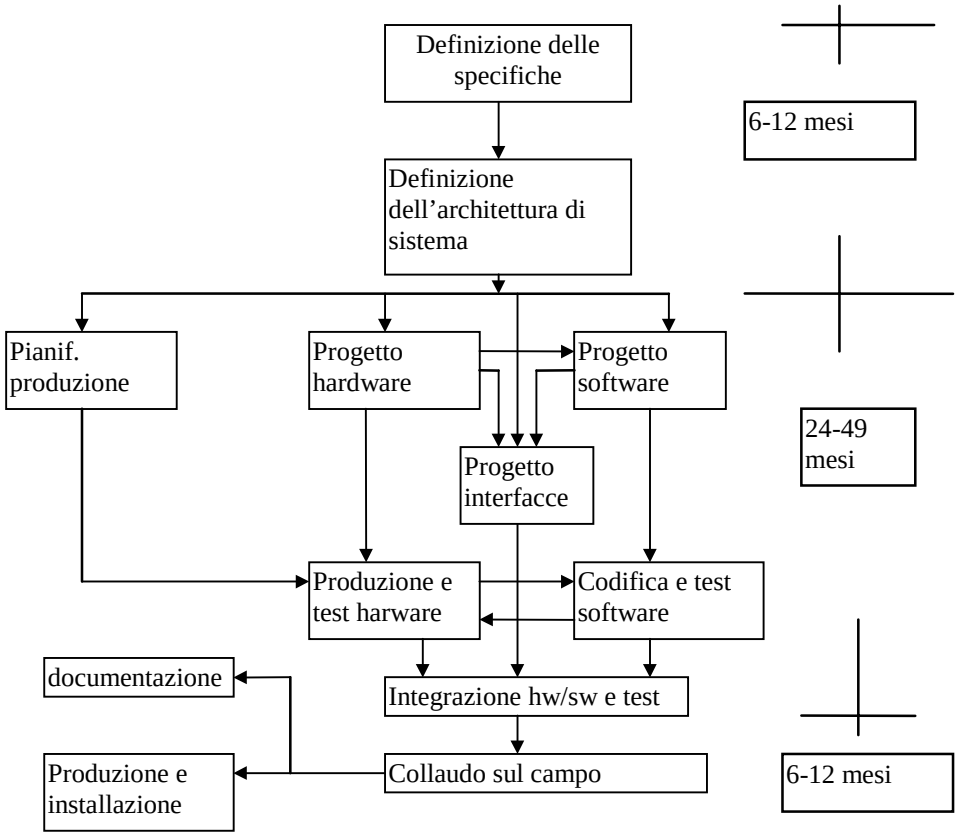
Idealmente, il progetto di un *sistema digitale dedicato a un'applicazione* (in molti casi, soggetto a vincoli di *tempo reale*) - quello che viene abitualmente indicato come un *sistema embedded* - dovrebbe essere supportato da un insieme di strumenti CAD che consentano di svolgere in modo totalmente o parzialmente automatizzato le seguenti attività :

- descrivere in modo formale *la specifica del sistema* al livello più elevato possibile: nel caso più generale - di un sistema digitale composto in parte da dispositivi standard programmabili (microprocessori o microcontrollori) con le relative memorie e interfacce di I/O, in parte da dispositivi progettati "ad hoc" (i cosiddetti ASIC) si

vorrebbe portare la specifica *a monte dell'assegnazione dei diversi compiti a componenti hardware o software* (a monte del "partizionamento" hardware-software);

- operare una verifica della correttezza/completezza delle specifiche o mediante simulazione (la tecnica attualmente più realistica) o mediante verifica formale (*dimostrazione di correttezza*);
- utilizzare strumenti di sintesi automatica (eventualmente con l'intervento del progettista nei punti-chiave delle scelte) per passare dalle specifiche ai circuiti logici, seguendo una tecnica "top down";
- in parallelo ai passi successivi della sintesi guidare la definizione delle strategie di collaudo.

Il flusso del progetto per un sistema complesso può anche essere riassunto dalla seguente figura:



Figura

L'approccio tradizionale ha un insieme di svantaggi, quali:

- tempi lunghi di sviluppo del prototipo
- alti costi di progetto
- limitata esplorazione delle architetture alternative
- mancanza di sistematico riuso di blocchi già progettati
- mancanza di hardware-software co-design sistematico.

La tendenza attuale è verso metodologie sistematiche che superino i problemi su elencati.

Nel seguito del corso si supporranno note le tecniche della *sintesi logica* su cui si basano gli attuali strumenti di CAD a livello *logico*: si richiameranno solo brevemente alcuni principi riguardanti modellazione e simulazione, indispensabili anche per le trattazioni successive. Ci si concentrerà dapprima sui problemi relativi al *collaudo* (e al progetto mirato alla *collaudabilità*), riferendosi soprattutto al livello logico e - per sistemi particolari - al livello funzionale, per concentrarsi quindi sulle problematiche poste *dalla sintesi e dalla specifica ad alto livello* di astrazione, che consentono al progettista di fornire una descrizione *algoritmica* del proprio sistema ed ottenere una sintesi (opportunamente ottimizzata) a livello di registri e operatori funzionali. Si passerà poi dal livello funzionale a quello di *sistema*, esaminando alcune tecniche per la specifica formale a tale livello (e alcune problematiche inerenti a tale fase), affrontando infine gli aspetti essenziali di quello che viene chiamato oggi "*hardware-software co-design*", cioè il progetto di un sistema dedicato realizzato in parte mediante dispositivi digitali progettati ad hoc, in parte mediante microprocessori o microcontrollori standard per i quali si devono realizzare specifici moduli software e che devono interfacciarsi correttamente coi moduli hardware prima citati.

LIVELLI D'ASTRAZIONE

Prima di definire le diverse fasi del processo di sviluppo e quindi i diversi livelli d'astrazione a cui viene descritto il circuito, è fondamentale introdurre i tre diversi domini di rappresentazione del circuito. Gli assi del diagramma di figura 5 rappresentano i tre domini: funzionale, strutturale e fisico [15]. Lungo gli assi sono indicati i differenti livelli di descrizione del dominio: i punti lontani dal centro indicano i livelli di descrizione più astratti mentre il centro degli assi rappresenta il caso in cui il circuito è completamente specificato.

Nel dominio *funzionale* si è interessati solo a ciò che il circuito compie e non a come è realizzato. Il circuito è considerato come una scatola nera fornita di ingressi e uscite e di un'insieme di funzioni che descrivono il comportamento di ogni uscita. Oltre alla descrizione del comportamento di ogni uscita vengono anche riportate la descrizione dell'interfaccia e dei vincoli tecnologici imposti al progetto. La descrizione dell'interfaccia definisce le porte d'ingresso e d'uscita e le relazioni temporali tra i segnali presenti a queste porte.

La rappresentazione *strutturale* è intermedia rispetto alle altre due rappresentazioni. La corrispondenza fra la descrizione funzionale e quella strutturale è di uno a molti per un dato insieme di componenti e sotto predeterminati vincoli tecnologici.

La rappresentazione *fisica* ignora, quanto più è possibile, ciò che il circuito realizza. Questa rappresentazione specifica posizione nello spazio o su silicio dei diversi componenti. Il passaggio dal dominio strutturale a quello fisico viene in genere svolto in due passi. Il primo passo dispone in maniera approssimata i diversi elementi circuitali curando il posizionamento delle interconnessioni. Nel secondo passo si sostituiscono gli elementi circuitali con la corrispondente rappresentazione fisica.

Se si sovrappongono ai tre assi di figura 5 delle circonferenze centrate nel centro degli assi si otterranno i diversi livelli di descrizione del circuito e in particolare le diverse fasi di realizzazione in un flusso *top-down* (se si parte dalla circonferenza più esterna per arrivare a quella più interna altrimenti se il verso è quello opposto il flusso è *bottom-up*). La circonferenza più esterna indica il livello di sistema ed è seguita dai livelli: algoritmico-funzionale, microarchitetturale (*Register Transfer Level*, *RTL*), logico e di circuito (tabella 2) [15].

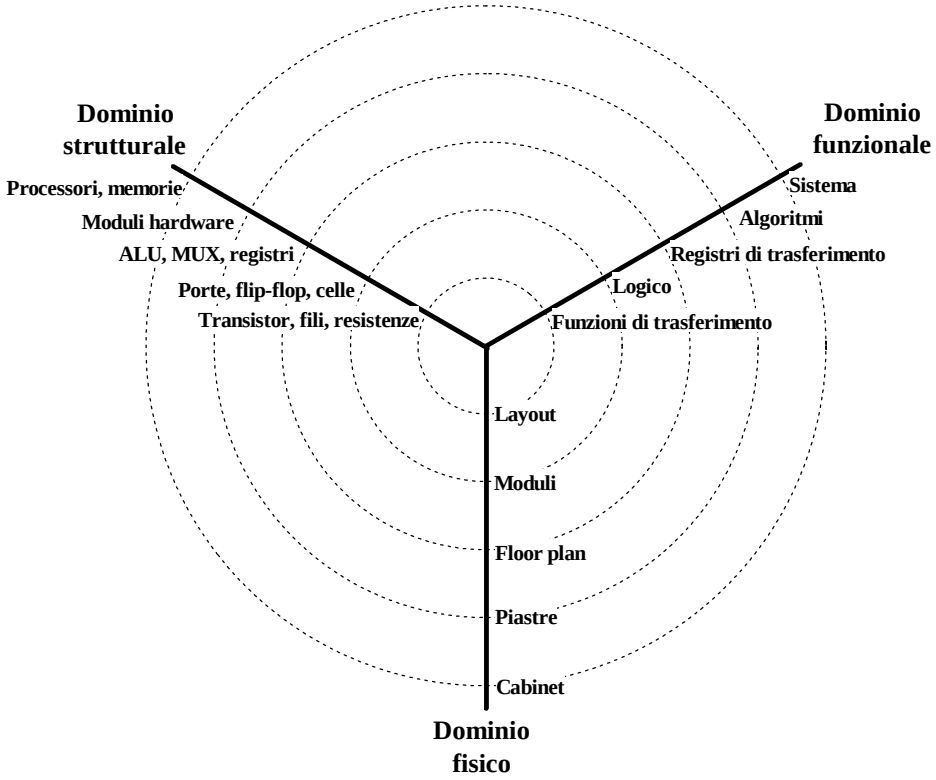


Figura 5{Gajski}:Domini di rappresentazione di un circuito. Nel dominio *funzionale* si è interessati solo a ciò che il circuito compie e non a come è costruito. La rappresentazione *strutturale* è intermedia rispetto alle altre due rappresentazioni. La rappresentazione *fisica* ignora, quanto più è possibile, ciò che il circuito realizza.

A livello di *circuito* la descrizione funzionale è realizzata attraverso l'uso di funzioni di trasferimento e di diagrammi temporali. Nella descrizione strutturale gli elementi base sono transistor, resistenze e fili di interconnessione. Nel domino fisico la rappresentazione è quella tipica di un layout di un circuito integrato.

A livello *logico* la funzionalità è espressa da espressioni booleane, la struttura ha come elementi base le porte logiche, i flip-flop e le celle, mentre la descrizione fisica realizza il posizionamento delle celle e dei moduli nello spazio.

A livello *Register Transfer* il comportamento è espresso da registri di trasferimento e da macchine a stati finiti (FSM - Finite State Machine), la struttura ha come componenti base ALU, multiplexer, registri, microcontrollori e micromemorie, mentre la descrizione fisica realizza il posizionamento dei diversi componenti nello spazio.

Livelli	Domini		
	Funzionale	**Strutturale**	**Fisico**
Sistema	Specifiche	CPU, memorie, switch	Cabinet
Algoritmo	Algoritmo	Moduli hardware	Schede
Register Transfer	Registri di trasferimento, FSM	ALU, MUX, registri, microstore	Floor plan
Logico	Equazioni booleane, FSM	Porte logiche, Flip-Flop, Celle	Celle, module plan
Circuito	Funzioni di trasferimento, diagrammi temporali	Transistor, resistenze e fili	Layout

Tabella 2{ferra10}: Incrocio tra domini di rappresentazione e livelli di descrizione

A livello *algoritmico* la funzionalità è descritta da un algoritmo che definisce le strutture dati e la sequenza delle manipolazioni che devono essere eseguite. Le variabili e le strutture dati usate dall'algoritmo non corrispondono necessariamente a registri o a memorie e così le operazioni non corrispondono, in genere, a blocchi funzionali o ad unità di controllo. L'algoritmo può essere espresso formalmente attraverso specifici linguaggi per la descrizione dello hardware (*Hardware Description Language*, *HDL*) come il VHDL [18]. La descrizione strutturale ha gli stessi elementi base di quella microarchitetturale solo che a questo livello gli elementi sono raggruppati in linee di trasferimento dati, unità di controllo e unità di immagazzinamento. Inoltre viene posto maggiore accento alla sincronizzazione e alla comunicazione fra i diversi componenti piuttosto che alla loro realizzazione. Le entità principali a livello fisico sono le schede.

A livello di *sistema* la descrizione funzionale specifica le prestazioni e i vincoli generali del circuito, tralasciando aspetti quali ad esempio il tipo di dati manipolati e il tipo di algoritmi sfruttati. Gli elementi base della descrizione strutturale sono processori, memorie, unità di controllo, switch e bus. La descrizione fisica considera il posizionamento e la suddivisione del circuito a livello di gruppi di schede o di armadi.

Il resto di questa sezione è dedicato alla descrizione delle principali architetture e dei modelli dei dispositivi elettronici digitali utilizzate nella progettazione a livello logico, RT e di sistema.

Capitolo 3

INTRODUZIONE ALLA PROGETTAZIONE LOW-POWER

INTRODUZIONE

Le problematiche legate al consumo di potenza hanno acquisito notevole rilevanza negli ultimi anni a causa della crescente richiesta di applicazioni elettroniche portatili e di sistemi complessi ad elevate prestazioni. La minimizzazione della potenza dissipata, dettata nella prima categoria di prodotti dalla necessità di prolungare il tempo di autonomia delle batterie, è mossa in generale dall'esigenza di ridurre i costi di realizzazione dei circuiti, dovuti essenzialmente al raffreddamento e al *packaging*, e di aumentare l'affidabilità dei sistemi. Il consumo energetico è divenuto uno dei vincoli più stringenti durante il progetto di circuiti a larga scala di integrazione, giustificando così gli enormi sforzi di ricerca estesi a tutti i livelli di astrazione che caratterizzano lo sviluppo dell'applicazione.

SORGENTI DEL CONSUMO DI POTENZA

A determinare la dissipazione di potenza in un circuito CMOS contribuiscono le seguenti quattro componenti [Weste]:

- la **corrente di dispersione** *(leakage current)* che è controllata, in massima parte, dal processo di fabbricazione tecnologico, e può essere ulteriormente suddivisa nelle due componenti:
  - corrente di polarizzazione inversa nei diodi parassiti formati dalle due diffusioni di sorgente (*source*) e di pozzo (*drain*) con la regione di substrato (*bulk*) nel *transistor* MOS;
  - correnti sottosoglia dovute all'inversione di carica che esiste per tensioni di pilotaggio *gate-source* inferiori alla tensione di soglia.
- la **corrente di standby** *(standby current)*, ovvero la corrente continua che fluisce dalla alimentazione verso massa quando il dispositivo è in uno stato stabile.
- la **corrente di corto circuito** *(short-circuit current)* dovuta al percorso continuo che si crea tra alimentazione e massa durante una transizione dell'uscita.
- la **corrente capacitiva** *(capacitance current)*, detta anche corrente di commutazione, legata al flusso di carica e scarica della capacità di carico durante le transizioni logiche dell'uscita.

Si è soliti indicare con il termine *dissipazione di potenza statica* la somma delle dissipazioni dovute alla corrente di dispersione e alla corrente di *standby*. Queste ultime sono presenti, per esempio, in un invertitore pseudo-nMOS quando sia il *transistor* nMOS che il *transistor* pMOS sono nello stato ON, oppure quando il *drain* di un *transistor* nMOS pilota il *gate* di un altro *transistor* nMOS in una *pass-transistor logic*. Si consideri ad esempio l'invertitore pseudo-nMOS rappresentato in figura 1.6.

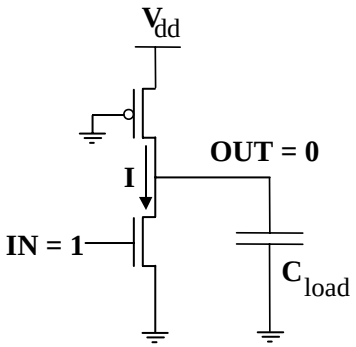


Figura 1.6{Fig_pseudonmos}. Dissipazione statica in un invertitore pseudo-nMOS.

In questa realizzazione il *transistor* pMOS è sempre attivo. Quando l'uscita della porta è a livello logico basso esiste un percorso a bassa impedenza tra alimentazione e massa e circola una corrente statica che può dare origine ad una dissipazione considerevole. Per questa ragione famiglie logiche di questo tipo non dovrebbero essere utilizzate in progetti orientati al basso consumo di potenza.

La componente legata ai diodi polarizzati inversamente è proporzionale all'area della regione di diffusione di *drain* e tipicamente ha un valore dell'ordine di 1-5pA/μm^2 a 25°C [Brod95b].

La corrente sottosoglia, che costituisce il contributo maggiore alla corrente di dispersione, è esprimibile attraverso la relazione seguente:

(1.1{Equ_a1}) $I_{DS} = Ke^{(V_{GS}-V_t)/(\eta V_T)}\left(1-e^{V_{DS}/V_T}\right)$,

dove K è un fattore legato alla tecnologia, proporzionale al rapporto (W/L) tra la larghezza e la lunghezza di canale, η è un parametro che vale circa 1.5, V_T è la tensione termica, V_{DS} la tensione *drain-source* e V_t la tensione di soglia del dispositivo. Questa relazione, valida per dispositivi a canale lungo, mette in evidenza che la corrente sottosoglia aumenta linearmente con il rapporto (W/L), e diminuisce esponenzialmente con $(V_{GS} - V_t)$. La dipendenza di I_{DS} da V_{DS} scompare per valori della tensione *drain-source* sufficientemente elevati, tipicamente per $V_{DS} > 0.1V$. Con la riduzione della tensione di soglia queste correnti potrebbero diventare più pronunciate.

Con il termine *dissipazione di potenza dinamica* si intende, invece, la somma delle dissipazioni dovute alla corrente di corto circuito ed alla corrente capacitiva, presenti solo durante le transizioni logiche dell'uscita.

Il consumo di potenza dovuto alla corrente di corto circuito per un invertitore è proporzionale ai tempi di salita e di discesa degli ingressi, al carico ed alle dimensioni dei dispositivi che implementano la funzione logica. La massima corrente di corto circuito fluisce quando il carico è assente, e diminuisce con l'aumentare del carico stesso. Se il dimensionamento dei componenti è tale da rendere circa uguali i tempi di salita e discesa degli ingressi e delle uscite, il consumo di potenza può diminuire a circa il 15% della totale potenza dinamica [Pedr96]. Se però si progetta per elevate prestazioni, con dispositivi di grosse dimensioni che pilotano carichi piccoli, e con tempi di commutazione lunghi degli ingressi, tale percentuale può aumentare in modo significativo.

Con riferimento all'invertitore CMOS mostrato in figura 1.7 l'intervallo di tempo durante il quale si instaura il percorso a bassa impedenza è quello in cui è verificata la relazione:

(1.2{Equ_a2}) $V_{tn} < V_{in} < V_{dd} - |V_{tp}|$.

Risulta allora evidente l'influenza della pendenza dei segnali di ingresso sul consumo di corto circuito: più lenta sarà la transizione di V_{in}, più a lungo vi sarà conduzione tra V_{dd} e massa.

La causa dominante della dissipazione di potenza nei circuiti CMOS è la carica e scarica delle capacità dei nodi, chiamata anche dissipazione di potenza capacitiva. Si faccia riferimento ancora all'invertitore di figura 1.7 e si consideri il caso di un ciclo completo di operazioni con due transizioni sul segnale di ingresso. Quando l'ingresso commuta da 1 a 0, viene generata una corrente (I_{char}) e caricata la capacità C_{load}. Si può facilmente ricavare che l'energia, misurata in joule, fornita dall'alimentazione è:

(1.3{Equ_a3}) $E = C_{load} V_{dd}^{\,2}$,

dove V_{dd} è la tensione di alimentazione.

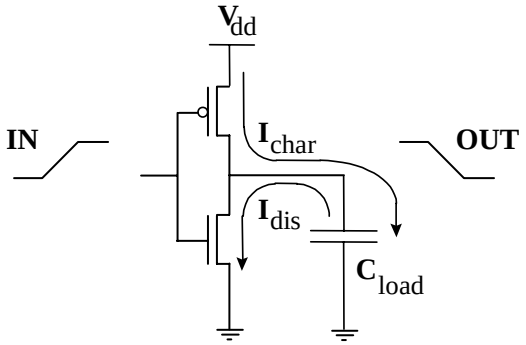


Figura 1.7{Fig_carica_scarica}. Flusso di corrente in un invertitore CMOS.

Metà di questa energia è dissipata attraverso il *transistor* pMOS, a causa della corrente I_{char}. L'altra metà è accumulata nella capacità C_{load}. Quando il segnale di ingresso cambia a 1, circola una corrente I_{dis} e la capacità C_{load} si scarica. L'energia immagazzinata in C_{load} durante la precedente transizione dell'ingresso viene ora dissipata attraverso il *transistor* nMOS; l'alimentazione non fornisce ulteriore energia. Perciò, l'energia media erogata dall'alimentazione per una coppia di transizioni dell'uscita, da 0 a 1 e da 1 a 0, è:

(1.4{Equ_a4}) $E_{avg} = \frac{1}{2} C_{load} V_{dd}^{\,2}$.

Assumendo che l'uscita dell'invertitore commuti ad ogni ciclo, il consumo di potenza medio dovuto alla carica e scarica della capacità C_{load} è dato dall'equazione:

(1.5{Equ_a5}) $P_{av} = \frac{1}{2} C_{load} V_{dd}^{\,2} f$,

dove f è la frequenza operativa, che potrebbe essere la frequenza di *clock*, $f_{clock} = 1/T_{clock}$, in un sistema sincrono.

Tenuto conto che non sempre le uscite delle porte commutano, l'espressione precedente può essere modificata nel seguente modo:

(1.6{Equ_Pav_switching}) $P_{av} = \frac{1}{2} C_{load} V_{dd}^{\ 2} f \alpha$,

dove α, detta attività di commutazione (*switching activity*), rappresenta il numero di volte che la capacità C_{load} è caricata o scaricata durante un ciclo.

Il contributo di questa componente alla potenza totale dissipata dipende dal tipo di circuito. Per circuiti di tipo flusso-dati (*data-path*) come moltiplicatori, sommatori, DSP, ecc.., essa influisce per circa il 90% della dissipazione di potenza complessiva, tuttavia anche in applicazioni differenti risulta essere dominante rispetto ai termini analizzati in precedenza.

IL PROGETTO A BASSA POTENZA

L'equazione (1.6) mette in evidenza i tre gradi di libertà a disposizione per ridurre la dissipazione di potenza nell'ipotesi che la frequenza operativa sia fissata: tensione di alimentazione, capacità fisica, e *switching activity*. L'ottimizzazione in potenza consiste allora nel tentativo di ridurre uno o più di questi fattori tenendo però in considerazione il fatto che, non essendo ortogonali tra loro, non possono essere fissati in modo indipendente l'uno dall'altro. Nelle sezioni seguenti verranno illustrati tali fattori in dettaglio, descrivendo la dipendenza di ognuno di essi dalle strategie realizzative e sottolineandone l'interazione reciproca.

Tensione di alimentazione

La potenza dissipata subisce un notevole miglioramento quando si riduce la tensione di alimentazione, a causa della dipendenza quadratica da tale parametro. Inoltre valori contenuti di V_{dd} migliorano l'affidabilità dei dispositivi, limitando il riscaldamento dei portatori nel canale e preservando l'integrità degli isolanti [Ricc95]. Purtroppo questa soluzione, applicabile durante le diverse fasi di progetto in quanto indipendente dalla logica e dalla tecnologia del circuito, incide negativamente sulla velocità del dispositivo e potrebbe far nascere problemi di compatibilità.

Al diminuire della tensione di alimentazione, il ritardo del circuito aumenta degradando le prestazioni del sistema (figura 1.8).

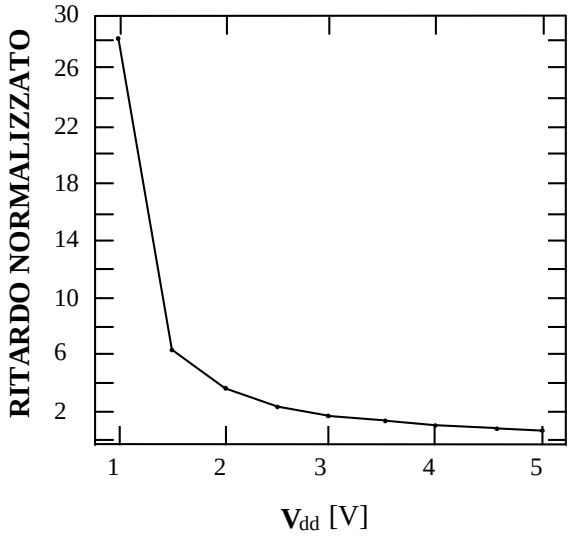


Figura 1.8{Fig_delay_Vdd}. Ritardo normalizzato in funzione di Vdd.

In prima analisi, facendo riferimento ancora all'invertitore CMOS, la corrente che fluisce nel dispositivo può essere espressa dalla relazione:

$$(1.7\{Equ\_a7\}) \quad I_{char} = \frac{\mu C_{ox}}{2}\left(\frac{W}{L}\right)(V_{dd} - V_t)^2 ,$$

e quindi il ritardo del circuito è dell'ordine di:

$$(1.8\{Equ\_ritTd\}) \quad \{Equ\_ritardo\_Td\} \quad T_d = \frac{C_{load} \times V_{dd}}{I_{char}} = \frac{C_{load} \times V_{dd}}{\frac{\mu C_{ox}}{2}\left(\frac{W}{L}\right)(V_{dd} - V_t)^2} .$$

Così per $V_{dd} >> V_t$ il ritardo aumenta linearmente con il diminuire della tensione e, per soddisfare i requisiti di prestazione del sistema, è necessario adottare alcune tecniche a livello tecnologico o a livello architetturale per compensare questo effetto. Per tensioni prossime a V_t la velocità del circuito viene penalizzata in maniera significativa, impedendo di fatto la possibilità di alimentare il sistema con tensioni al di sotto di $2V_t$.

Le prestazioni non sono però il solo fattore che limitano la scelta della tensione di alimentazione; ad esempio l'utilizzo di livelli non standard potrebbe dare origine a problemi di compatibilità con altre sotto parti del sistema che si vuole realizzare.

Capacità fisica

La diminuzione delle capacità fisiche può essere ottenuta fondamentalmente riducendo le dimensioni dei dispositivi e delle interconnessioni. Per i dispositivi il contributo maggiore deriva dalle capacità di giunzione e di *gate* (figura 1.9).

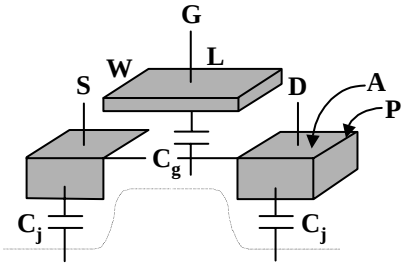


Figura 1.9{Fig_capac_MOS}. Capacità associate al dispositivo MOS.

La capacità associata allo spessore dell'ossido di *gate* del *transistor* è solitamente la maggiore delle due e può essere espressa mediante la ben nota relazione:

$$(1.9\{Equ\_a9\}) \quad C_g = WLC_{ox} = WL\frac{\varepsilon_{ox}}{t_{ox}} .$$

Le capacità dovute alle giunzioni di *source* e di *drain* variano in modo non lineare con la tensione applicata ai capi della giunzione e dipendono sia dall'area che dal perimetro della due regioni di diffusione:

(1.10{Equ_a10}) $$C_j(V) = AC_{j0}\left(1-\frac{V}{\phi_0}\right)^{-m} + PC_{jsw0}\left(1-\frac{V}{\phi_0}\right)^{-m},$$

dove A e P sono, rispettivamente, l'area e il perimetro della regione considerata (*source* o *drain*), C_{j0} e C_{jsw0} i valori di capacità all'equilibrio, ϕ_0 il potenziale di barriera della giunzione ed m una costante che dipende dal profilo di drogaggio della giunzione.

A determinare la capacità complessiva di una connessione contribuiscono, invece, sia la capacità tra ciascun livello di metallizzazione ed il substrato, che le capacità mutue tra i livelli stessi (figura 1.10).

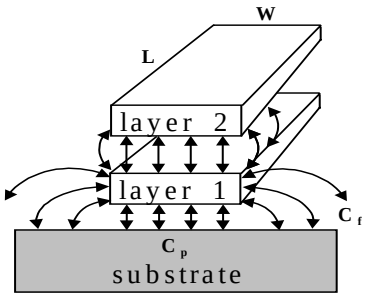


Figura 1.10{Fig_Cap_fili}. Capacità associate alle interconnessioni.

Indicando con C_p la capacità per unità di area tra due connessioni parallele e con C_f la capacità per unità di lunghezza dovuta agli effetti di bordo in prima analisi vale la relazione:

(1.11{Equ_a11}) $$C_{wire} = WLC_p + 2(W+L)C_f .$$

La componente predominante è quella parallela mentre quella dovuta agli effetti di bordo diventa significativa solo nel caso in cui la larghezza del collegamento diventa stretta rispetto allo spessore.

Così come non è possibile scegliere la tensione di alimentazione di un circuito sulla base di considerazioni legate alla sola dissipazione, anche la minimizzazione delle capacità fisiche non può essere fatta in modo del tutto indipendente. Ad esempio se da un lato l'utilizzo di *transistor* a dimensioni ridotte consente di ottenere dei vantaggi in termini di capacità fisica, dall'altro la velocità del circuito viene penalizzata poiché minore è la corrente che questi dispositivi sono in grado di fornire.

Attività di commutazione

In accordo con la relazione (1.6), oltre alla tensione di alimentazione e alla capacità fisica, il terzo parametro fondamentale che contribuisce a determinare la dissipazione di potenza è l'attività di commutazione α o *switching activity* del circuito. Mentre la frequenza operativa f individua la periodicità media con cui arrivano i dati in ingresso, α determina il numero medio di transizioni che si verificano ad ogni arrivo.

L'attività dei dati α può essere combinata con la capacità fisica C per ottenere una *capacità effettiva* $C_{eff} = \alpha C/2$ che rappresenta la capacità media caricata in ogni periodo $1/f$. Ciò in ragione del fatto che né la capacità fisica né l'attività di commutazione da sole sono responsabili della potenza dissipata in un circuito CMOS:

(1.12{Equ_a12}) $P_{av} = \frac{1}{2}\alpha C V_{dd}^2 f = C_{eff} V_{dd}^2 f$.

Estendendo la relazione (1.6) a tutti i nodi di un circuito, e ricordando che per la linearità dell'operatore di media il valore medio di una somma è la somma dei valori medi, si ricava un'espressione che permette di calcolare la potenza dissipata complessiva:

(1.13{Equ_Pav_activity}) $P_{av} = \frac{1}{2T_c} V_{dd}^2 \sum_{i=1}^{n} C_i \alpha(x_i)$,

dove $\alpha(x_i)$ è la *switching activity* del segnale al nodo x_i, n è il numero di nodi del circuito e T_c è il periodo di *clock*.

Note le capacità C_i nei nodi, il vero problema nel calcolo della (1.13) è la determinazione delle *switching activity* $\alpha(x_i)$. Tali valori, come verrà chiarito nel capitolo successivo, sono difficili da calcolare perché dipendenti da un certo numero di fattori legati ai parametri circuitali e alla tecnologia che non sono facilmente disponibili o caratterizzati con precisione. Alcuni di questi fattori sono [Pedr96]:

- modello di ritardo;
- funzione logica;
- proprietà statistiche degli ingressi;
- stile logico;
- topologia del circuito.

TECNICHE DI OTTIMIZZAZIONE DELLA POTENZA

Si analizzeranno in questa sezione alcune fra le più significative strategie adottate in fase di progetto per limitare i consumi, pur mantenendo inalterate le prestazioni del circuito. Si affronteranno le problematiche connesse ai diversi livelli di astrazione proponendo per ognuno di essi soluzioni che saranno, in generale, il compromesso fra le opposte esigenze di ridurre la dissipazione e di mantenere i vincoli di area occupata e di velocità di elaborazione.

A livello fisico-circuitale, per esempio, si possono condizionare gli algoritmi di collegamento e di posizionamento dei dispositivi sul silicio, la dimensione dei singoli *transistor* potrebbe essere ridotta lungo i percorsi non critici del circuito oppure risultati consistenti possono essere ottenuti attraverso processi di *scaling* tecnologico.

A livello logico gli stati simbolici di una macchina a stati finiti (FSM) possono essere codificati in modo tale da minimizzare il numero di bit che cambiano nella logica combinatoria per le transizioni di stato più probabili; porzioni di circuito che non contribuiscono al calcolo del valore attuale dell'uscita potrebbero essere disabilitate; i blocchi di logica combinatoria possono inoltre essere ottimizzati tenendo conto, nelle funzioni di costo, del loro impatto sul consumo complessivo del circuito.

Spostandosi verso il livello RT, l'adozione di particolari architetture (parallelismo e *pipelining*) consentono di aumentare la velocità del circuito e quindi di poter ridurre la tensione di alimentazione, con notevole risparmio sulla dissipazione, mantenendo costante il numero di operazioni compiute nell'unità di tempo dall'intero sistema (*throughput*). Anche le procedure di

scheduling, allocazione ed assegnamento delle risorse, la codifica dei dati utilizzata ed il partizionamento del circuito possono contribuire a limitare il consumo del *chip* che si intende realizzare.

Le soluzioni adottate al più alto livello di astrazione, quello comportamentale, come ad esempio le trasformazioni che aumentano la concorrenza o la sostituzione di operazioni, possono influire anch'esse in modo determinante sul risultato dell'implementazione finale, ed è necessario tenerne conto nel corso della progettazione.

In generale le maggiori riduzioni della potenza dissipata sono ottenute nelle fasi iniziali del processo di progettazione, ovvero a livello architetturale e comportamentale. Le tecniche elencate, ed illustrate in maniera più dettagliata nelle successive sezioni, sono tuttavia ortogonali tra loro e quindi possono, ed in linea di principio devono, essere impiegate simultaneamente per ottenere risultati apprezzabili.

Livello circuitale

Si considerano in questa sezione le tecniche di riduzione della dissipazione applicabili al livello più basso dell'intero processo di progettazione, quello di *layout*. In questa fase tecniche di ottimizzazione sono applicabili al dimensionamento dei transistori, al posizionamento dei dispositivi sul silicio e agli algoritmi di collegamento.

Riduzione delle soglie dei dispositivi

Si è osservato come i ritardi aumentino con l'approssimarsi della tensione di alimentazione alle soglie dei dispositivi. Dato che l'obbiettivo è ridurre il consumo di potenza mantenendo costante il *throughput*, è necessario compensare questo ritardo adottando opportune soluzioni architetturali come parallelismo e *pipelining*. Questi metodi verranno illustrati in maniera dettagliata quando si affronterà l'ottimizzazione a livello RT.

Un'altra possibilità per ridurre V_{dd} a parità di *throughput* è quella di utilizzare *transistor* MOS con tensione di soglia più bassa. Dall'espressione (1.8) si vede come, mantenendo inalterata la differenza $(V_{dd} - V_t)$, sia possibile ridurre V_{dd} senza perdita di prestazioni. La figura 1.11 illustra l'andamento del ritardo normalizzato in funzione della tensione di soglia, per diversi valori della tensione di alimentazione. Si può notare come, per valori di V_t al di sotto di 0,5V, il ritardo aumenti in modo contenuto al diminuire di V_{dd}.

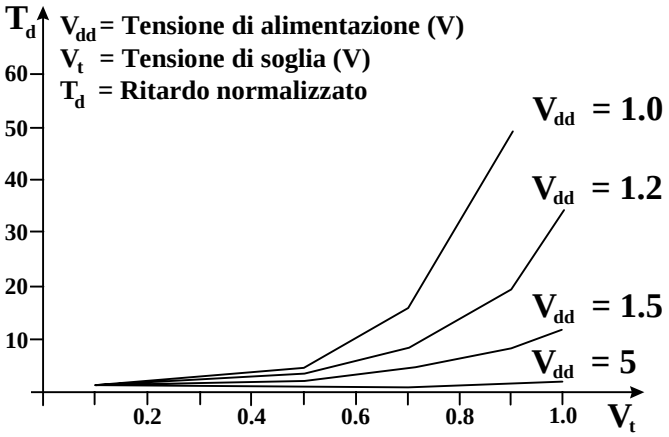


Figura 1.11{Fig_Td_Vt}. Effetto della riduzione delle soglie sul ritardo, per diversi valori di Vdd.

Il limite alla riduzione della tensione di soglia è dato comunque dalla necessità di mantenere dei margini di rumore accettabili e una dissipazione di potenza statica limitata, visto che le correnti sottosoglia aumentano con il diminuire di V_t.

Partizionamento del circuito

Il partizionamento del progetto è indispensabile quando si vogliono sintetizzare circuiti complessi; la suddivisione in blocchi consente infatti di analizzare ed ottimizzare separatamente le diverse parti. In generale però le capacità esterne al blocco sono molto maggiori (da uno a due ordini di grandezza) rispetto a quelle che si trovano all'interno del blocco [Pedr96]. È essenziale quindi sviluppare delle tecniche di partizionamento che mantengano i cammini critici, con elevata attività di commutazione, il più possibile all'interno dello stesso blocco.

Ottimizzazione di place and route

Con i termini *"place and route"* si è soliti indicare la fase del progetto in cui si stabiliscono quali posizioni assegnare ai dispositivi sul silicio e quali percorsi seguire per la loro interconnessione. È evidente che la disposizione dei vari componenti influisce in maniera determinante sui carichi capacitivi delle porte quindi, a questo livello, l'ottimizzazione consiste nel posizionare vicine tra loro le porte che commutano con maggiore frequenza, in modo che la capacità dei collegamenti pilotati risulti minima, e allontanando progressivamente le porte che commutano più raramente, dato che per queste un aumento del carico capacitivo presenta un impatto minore sul consumo globale. In [Chao94] viene descritto un metodo basato sulla generazione di curve di forma parametrizzate, in modo da tenere in considerazione il dato relativo alla dissipazione di potenza. Queste stesse curve possono poi essere utilizzate per determinare la dimensione ottima dei moduli e la loro posizione in base ai vincoli di area, tempo e potenza impostati dal progettista.

Dimensionamento delle porte e delle connessioni

La dimensione dei *transistor* può avere un impatto notevole sui ritardi e sulla dissipazione del circuito. All'aumentare del rapporto W/L aumenta infatti la corrente che il dispositivo è in grado di fornire e quindi, in accordo con la (1.8), il ritardo della porta diminuisce, mentre aumenta la dissipazione a causa dell'incremento delle capacità parassite. Si pone allora il problema di determinare, dato un certo vincolo temporale, la dimensione ottimale dei *transistor* che minimizza il consumo di potenza. In [Brod95b] si mostra che non sempre questo ottimo coincide con il valore minimo del rapporto W/L, poiché è necessario considerare anche il contributo dato dalle interconnessioni alla capacità di carico totale della porta.

Si consideri ad esempio il semplice circuito in figura 1.12, in cui il primo stadio pilota la capacità di gate del secondo stadio e la capacità parassita C_p dovuta al substrato e alle interconnessioni.

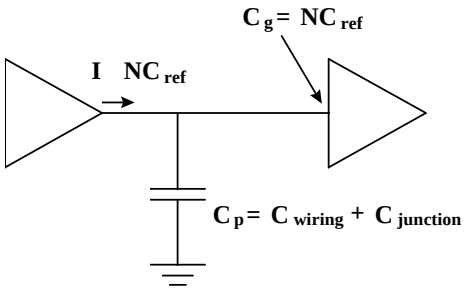


Figura 1.12{Fig_DimPorte}. Modello per l'analisi degli effetti del dimensionamento dei transistor.

Se la capacità di ingresso di entrambe le porte è data da *NCref*, dove *Cref* rappresenta la capacità del dispositivo MOS avente il più piccolo rapporto W/L permesso, allora il ritardo introdotto dalla prima porta, per una tensione di alimentazione *Vref* fissata, è dato dalla relazione seguente:

(1.14{Equ_RitDIM}) $$T_N = K\frac{\left(C_p + NC_{ref}\right)}{NC_{ref}}\frac{V_{ref}}{\left(V_{ref} - V_t\right)^2} = K\left(1+\frac{\alpha}{N}\right)\frac{V_{ref}}{\left(V_{ref} - V_t\right)^2},$$

dove α è il rapporto tra le capacità C_p e C_{ref}, e K è un termine indipendente dalle dimensioni del dispositivo e dalla tensione di alimentazione.

Fissata V_{ref}, l'aumento di velocità di un circuito il cui W/L sia N volte maggiore rispetto a quello di un circuito facente uso di *transistor* di dimensione minima (ovvero con $N = 1$) è dato dal fattore $(1+\alpha) / (1+\alpha / N)$. Poiché l'obbiettivo è quello di diminuire il consumo di potenza a parità di *throughput,* si può pensare di ridurre la tensione di alimentazione del dispositivo più veloce così da rendere uguali i ritardi nei due casi, e comparare quindi le prestazioni energetiche.

Assumendo che il ritardo sia inversamente proporzionale alla tensione di alimentazione, si ricava che il valore V_N di tensione con cui è necessario alimentare il circuito scalato, cosicché il ritardo introdotto sia uguale a quello del circuito di riferimento alimentato da V_{ref}, è dato dall'espressione:

(1.15{Equ_RitVno}) $$V_N = \frac{(1+\alpha/N)}{(1+\alpha)}V_{ref}\ .$$

In queste ipotesi, l'energia consumata dal primo stadio, in funzione di N, vale

(1.16{Equ_RitEn}) $$E(N) = (C_p + NC_{ref})V_N^{\ 2} = \frac{NC_{ref}(1+\alpha/N)^3 V_{ref}^{\ 2}}{(1+\alpha)^2}\ .$$

In figura 1.13 è riportato l'andamento dell'energia normalizzata $E(N)/E(N{=}1)$, in funzione di N, per diversi valori del parametro α.

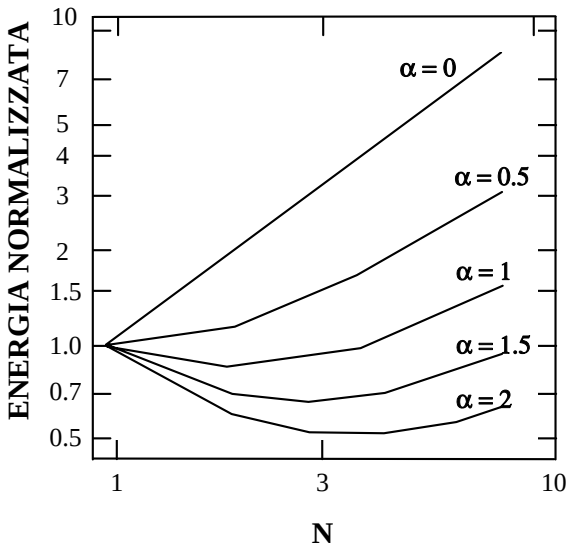


Figura 1.13{Fig_EnormN}. {Fig_Energia_N}Energia normalizzata in funzione della dimensione N del transistor al variare del contributo delle capacità parassite.

Il grafico mette in evidenza che per $\alpha = 0$, ovvero quando il contributo delle capacità parassite è nullo, la soluzione che utilizza il dispositivo di dimensione minima è anche quella più vantaggiosa per le applicazioni "*low power*". Viceversa, quando il contributo delle capacità parassite non è più trascurabile, nell'andamento dell'energia è presente un punto di minimo per $N > 1$ ed è questo il valore che dovrebbe essere utilizzato per determinare i rapporti di forma dei transistori. L'andamento della curva è giustificato dal fatto che inizialmente la diminuzione di tensione, resa possibile dalla riduzione del ritardo, è in grado di compensare l'incremento di capacità dovuto all'aumento di N. Oltre un certo valore, però, l'incremento di capacità è maggiore rispetto alla corrispondente riduzione di tensione e quindi la curva ritorna a salire.

L'analisi precedente è stata condotta nell'ipotesi che le capacità parassite siano indipendenti dalle dimensioni dei dispositivi, il che è verificato con buona approssimazione quando il maggiore contributo alla capacità di carico totale è dato dalle connessioni. In caso contrario, non è però possibile trascurare la dipendenza delle capacità di *drain* e di *source* da N e la dimensione ottimale dei *transistor* dal punto di vista della potenza dissipata ritorna ad essere quella minima [Brod95b].

Il problema del dimensionamento può essere affrontato non solo a livello del singolo transistore ma anche a livello di gate (*gate sizing*). La sola differenza è che, in quest'ultimo caso, tutti i rapporti di forma dei transistori sono calcolati simultaneamente anziché singolarmente.

Tipicamente gli algoritmi che si occupano di individuare la soluzione migliore assegnano una dimensione iniziale a ciascun dispositivo, sulla base delle precedenti considerazioni, quindi calcolano il ritardo introdotto da ciascuna porta rispetto a quello massimo consentito. In una seconda fase i sotto circuiti per cui questa differenza (*slack*) è maggiore di zero vengono processati e le dimensioni dei transistori ridotte finché lo *slack* diventa nullo, oppure finché non sono state raggiunte le dimensioni minime [Catt95]. È chiaro che, per sua natura, questo tipo di approccio consente di dimensionare una sola porta alla volta e quindi non rappresenta la migliore soluzione dal punto di vista computazionale.

Un'altra possibilità, presentata in [Berkel], è quella di fare ricorso a tecniche di programmazione lineare. Nel caso specifico, in particolare, si cerca di risolvere il seguente quesito: dato un circuito costituito da un insieme di celle, ed un limite massimo per il ritardo, determinare la forza di pilotaggio per ciascuna cella affinché il circuito soddisfi il vincolo temporale e, contemporaneamente, la dissipazione sia minima. È chiaro, infatti, che non tutte le celle richiedono la stessa capacità di pilotaggio; quelle che appartengono, ad esempio, a cammini critici e sono caratterizzate da un *fanout* elevato, dovrebbero essere in grado di pilotare carichi elevati. Viceversa celle che, pur avendo *fanout* elevato, non fanno parte di percorsi critici, non dovrebbero presentare questa caratteristica per diminuire il consumo di potenza. La soluzione proposta si basa su un modello semplificato per determinare il ritardo introdotto dalle celle e la relativa dissipazione; non si tiene in considerazione, ad esempio, né il contributo dato dalle correnti di corto circuito né il ritardo di salita e di discesa dei segnali di ingresso. Nonostante queste limitazioni forniscono buoni risultati in termini di risparmio energetico e consentono di processare un numero di *gate* dell'ordine delle diverse migliaia.

Livello logico

La sintesi logica è la fase del flusso di progetto in cui si compie il passo dalla descrizione architetturale (livello RT o di trasferimento fra registri) alla descrizione in termini di connessione tra porte logiche. Una sintesi logica che punti alla minimizzazione del consumo di potenza dovrà operare scelte atte a ridurre la capacità globalmente commutata; in particolare si cercherà di ridurre la frequenza delle transizioni dei nodi che pilotano grossi carichi capacitivi. Le tecniche di ottimizzazione a livello logico possono essere classificate in base a parametri differenti. Si distinguono infatti tecniche cosiddette "indipendenti dalla tecnologia", che vengono applicate al circuito in esame quando questo non è ancora stato "mappato" sulla libreria tecnologica scelta, da quelle invece "dipendenti dalla tecnologia" utilizzata; diversamente, in base alle caratteristiche del circuito che si vuole ottimizzare, si possono suddividere tra combinatorie e sequenziali. Le metodologie più significative, utilizzate per ridurre la potenza dissipata a livello di *gate*, possono allora essere suddivise in:

- Tecniche indipendenti dalla tecnologia per circuiti combinatori:
  - *"don't-care optimization"*;
  - ristrutturazione logica.
- Tecniche indipendenti dalla tecnologia per circuiti sequenziali:
  - assegnamento degli stati;
  - *"re-timing"*;
  - disabilitazione di porzioni del circuito.
- Tecniche dipendenti dalla tecnologia:
  - decomposizione tecnologica e *"mapping"*;
  - bilanciamento dei cammini.

Nel seguito si entrerà nel dettaglio delle sole tecniche per circuiti sequenziali, in quanto consentono di ottenere i risultati migliori dal punto di vista della dissipazione; nel caso dei circuiti combinatori in generale l'obbiettivo è quello di ottenere una rete di costo minimo dal punto di vista non solo dell'area e delle prestazioni, ma anche della potenza dissipata, e quindi è necessario modificare le funzioni di costo tradizionali introducendo metriche aggiuntive

capaci di stimare il consumo di un generico nodo *n* che realizza una data funzione logica *f*. Le tecniche dipendenti dalla tecnologia sono invece strettamente legate al tipo di libreria tecnologica che verrà utilizzata per la realizzazione del circuito. Per una trattazione più generale relativa a queste metodologie si rimanda a [Mazz97].

Assegnamento degli stati

L'assegnamento degli stati in una macchina a stati finiti (FSM), descritta solitamente mediante un diagramma delle transizioni di stato (STD), ha un notevole impatto sull'area, le prestazioni e la potenza dissipata dalla sua implementazione finale.

Individuare un assegnamento ottimo significa trovare una opportuna codifica binaria per tutti gli stati della FSM, in modo tale da minimizzare una data funzione di costo. È evidente che due stati S_i ed S_j fra i quali avviene una transizione dovrebbero essere codificati con configurazioni di bit adiacenti, se l'obbiettivo è quello di limitare la dissipazione, così da ridurre il numero di commutazioni nel registro di stato. Una possibile funzione di costo, presentata in [Pras93] è allora la seguente:

$$\sum_{i,j} W_{i,j} \cdot H\left(S_i, S_j\right), \qquad (1.17)$$

dove *Wi,j* è un coefficiente che tiene conto della probabilità di transizione *Si→Sj* ed *H(Si , Sj)* è la distanza di Hamming tra i codici binari assegnati ai due stati *Si* ed *Sj*. Nel lavoro originale si utilizza come coefficiente *Wi, j* la probabilità di transizione di stato condizionata

$$p_{i,j} = prob\left(S_i | S_j\right), \qquad (1.18)$$

che è solo una approssimazione dell'esatta probabilità: bisogna infatti considerare anche la probabilità *prob(Si)* di occupare lo stato stabile *Si*. I pesi *Wi,j* sono allora espressi come:

$$P_{i,j} = prob\left(S_i\right) \cdot prob\left(S_i | S_j\right), \qquad (1.19)$$

dove le probabilità di stato $prob(S_j)$ possono essere ricavate risolvendo le equazioni di Chapman-Kolmogorov.

Determinate le $P_{i,j}$ viene applicato l'algoritmo di assegnamento vero e proprio che, data la complessità del problema, si basa sulla tecnica di *"simulated-annealing"*. Inizialmente, fissato il numero di bit, i codici vengono assegnati casualmente agli stati; in una seconda fase si prova a scambiare i codici associati a due stati, scelti casualmente, oppure ad assegnare un nuovo codice, tra le configurazioni non ancora utilizzate, ad uno stato scelto sempre in maniera casuale. Queste mosse sono accettate sempre se la funzione di costo precedentemente definita diminuisce. In caso contrario sono accettate con una certa probabilità $e^{-|\delta|/T}$, dove δ è la variazione della funzione e T la temperatura di *annealing*.

Un difetto di questa tecnica è che viene minimizzata solo l'attività di commutazione delle linee di stato senza considerarne il carico capacitivo; inoltre viene completamente trascurato il consumo dovuto alla logica combinatoria che determina lo stato successivo e l'uscita della FSM.

Per tenere in considerazione l'influenza dell'assegnamento sull'area occupata, in [Maci96b] i coefficienti $W_{i,j}$ della funzione peso vengono modificati nel modo seguente:

(1.20{Equ_a20}) $W_{i,j} = \alpha \cdot w_{i,j}^{P} + (1-\alpha) \cdot w_{i,j}^{A}$,

dove α è un coefficiente compreso tra 0 e 1 $w_{i,j}^{P}$ e $w_{i,j}^{A}$ tengono conto uno del contributo alla dissipazione di potenza e l'altro dell'impatto sull'area occupata. Tanto più quest'ultima viene ridotta tanto minori saranno infatti le capacità, con l'aspettativa di migliorare quindi il bilancio energetico.

Re-timing

L'operazione di riposizionamento dei registri all'interno di un circuito sequenziale, senza modificarne il comportamento esterno, è chiamata *"re-timing"*.

Per mettere in evidenza l'impatto sul consumo di potenza si consideri la situazione mostrata in figura 1.14.

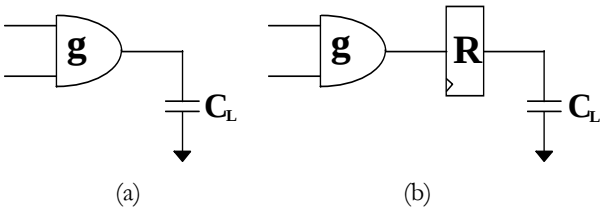


Figura 1.14{Fig_Retiming}. {Fig_retiming}Esempio di circuito prima (figura 1.14a) e dopo (figura 1.14b) il re-timing.

Se E_g è la *switching activity* dell'uscita della porta *g* e C_L la capacità di carico, la potenza dissipata per il circuito di figura 1.14a è proporzionale al prodotto $E_g C_L$. Nel secondo caso (figura 1.14b), detta E_R la *switching activity* all'uscita del registro e C_R la sua capacità in ingresso, la dissipazione è proporzionale a $E_g C_R + E_R C_L$. Dato che eventuali alee (*glitch*) presenti sull'uscita della porta *g* vengono filtrati dal registro *R* che può fare al più una transizione per ogni ciclo di *clock*, $E_R \leq E_g$ quindi può accadere che $E_g C_R + E_R C_L \leq E_g C_L$ se E_g e C_L sono sufficientemente grandi.

L'idea è quella di selezionare un insieme di nodi sulla base sia del numero di *glitch* in uscita che della probabilità di propagazione di questi ultimi ai nodi successivi, e quindi di inserire dei registri sulle loro uscite [Deva93]. Poiché un registro introduce di per se una dissipazione non trascurabile, i nodi candidati ad essere processati sono quelli che presentano un elevato *fanin* e *fanout*.

L'attività di commutazione E_{glitch}, dovuta ai *glitch*, viene stimata facendo la differenza $E_{real} - E_{zero}$ tra i valori ricavati nel corso di due simulazioni simboliche, una con modello di ritardo reale e l'altra con modello a ritardo nullo (si veda in proposito il paragrafo 2.3.2 del capitolo successivo).

Mediante un'altra simulazione simbolica viene poi determinata la probabilità $s_{i,j}$ che una transizione all'uscita del nodo *i* si propaghi all'ingresso del nodo *j* e, dato che l'obbiettivo è quello di minimizzare la dissipazione, questo valore è moltiplicato per la capacità di carico della porta a cui si riferisce. Una misura della riduzione di potenza ottenuta posizionando un registro all'uscita di un generico nodo *n* è data quindi dalla relazione seguente:

$$Power\_reduc_n = E^n_{glitch} \cdot \left(C_{L_n} + \sum_j^{fanout_n} \left(s_{j,n} \cdot C_{L_j} \right) \right), \quad (1.21\{Equ\_a21\})$$

dove C_{L_n} e C_{L_j} sono le capacità associate al nodo *n* e al nodo *j* rispettivamente.

Disabilitazione di porzioni del circuito

I sistemi digitali complessi contengono solitamente delle porzioni di logica che non compiono operazioni utili ad ogni ciclo di *clock*. È allora naturale pensare di disabilitare queste sotto reti, limitatamente ai cicli di *clock* in cui non sono in uso, al fine di ridurre la potenza dissipata. Così facendo si riduce l'attività di commutazione del circuito e quindi la componente dinamica della potenza.

Questa osservazione ha trovato applicazione nelle due tecniche che verranno descritte di seguito note con il nome di *"pre-computation"* e *"gated-clock"*.

Pre-computation

Per individuare quali porte del circuito disattivare durante determinati cicli di *clock* viene duplicata una porzione della logica con l'obbiettivo di calcolare, per alcune configurazioni d'ingresso, il valore dell'uscita del circuito con un ciclo di anticipo rispetto a quando è necessario. Il dato trovato in questo modo viene poi utilizzato nel periodo di *clock* successivo per ridurre l'attività di commutazione della logica. Infatti, conoscendo anticipatamente il valore dell'uscita, la rete originale può essere inibita per tutto il ciclo successivo, eliminando eventuali cariche delle capacità interne. D'altra parte, la dimensione della logica addizionale deve essere tenuta sotto controllo affinché il suo contributo al consumo di potenza non sia maggiore del risparmio che si può ottenere bloccando le commutazioni inutili all'interno del circuito. A questo scopo sono state proposte due architetture, caratterizzate da prestazioni differenti, che calcolano l'uscita solo in corrispondenza di un sotto insieme delle condizioni che si possono verificare in ingresso alla rete originale.

Si assuma che la struttura originale del circuito sia quella di figura 1.15.

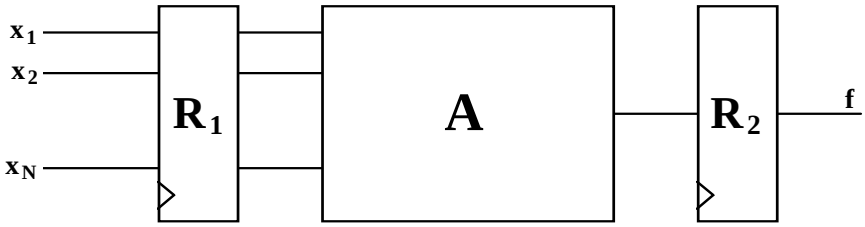


Figura 1.15{Fig_Precomp1}. Circuito originale.

Il blocco combinatorio A implementa una funzione booleana con *N* ingressi ed una singola uscita *f* ed ha i terminali connessi ai due registri *R*1 ed R_2.

La prima architettura proposta è mostrata in figura 1.16. Il valore dell'uscita *f* viene precalcolato mediante i due predittori ad *N* ingressi *g*1 e *g*2, il cui comportamento deve soddisfare il seguente vincolo:

$$\begin{cases} g_1 = 1 \Rightarrow f = 1 \\ g_2 = 1 \Rightarrow f = 0 \end{cases} \quad (1.22\{Equ\_Precomp\})$$

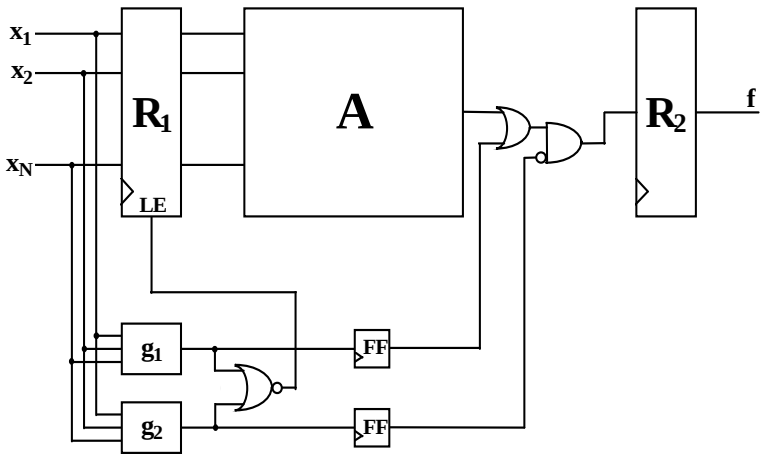


Figura 1.16{Fig_Precomp2}. Prima architettura di "pre-computation".

Se, durante l'*i*-esimo ciclo di *clock*, $g1$ o $g2$ valgono 1, il segnale *LE* disabilita il registro *R*1 forzando gli ingressi al blocco *A* a mantenere il loro valore anche al ciclo (*i*+1)-esimo. Ne deriva che il numero di transizioni sulle uscite delle porte interne al blocco *A* è nullo, mentre il valore corretto dell'uscita è fornito dai due *flip-flop* posti sulle uscite dei predittori.

Nella scelta di g_1 e g_2 una coppia di valori estremi è costituita da $g_1 = f$, $g_2 = f'$. Una scelta simile non porta ovviamente nessun beneficio in termini di risparmio energetico, mentre penalizza al massimo l'area occupata che diventa il doppio dell'area iniziale. L'obbiettivo è allora individuare due funzioni per cui la probabilità della loro somma ($g1 + g_2$) tenda ad 1 ma, contemporaneamente, la penalizzazione in termini di area dovuta alla loro implementazione sia limitata.

Per garantire che la logica addizionale sia meno complessa rispetto a quella che realizza la funzione *f* si fa in modo che l'uscita dei due blocchi $g1$ e $g2$ dipenda non da tutti gli *N* ingressi del blocco *A* ma da un insieme $k<N$. La struttura che ne deriva è riportata in figura 1.17, dove ancora una volta le funzioni di predizione $g1$ e $g2$ devono soddisfare i vincoli dati dalle equazioni (1.22). Si noti che in questo caso non viene mai calcolato il valore dell'uscita *f* in anticipo, ma viene determinato solo quando una parte degli ingressi al blocco combinatorio *A* può essere inibita.

Un semplice esempio di questa architettura applicata ad un circuito comparatore è mostrato in figura 1.18. L'uscita vale uno se $C > D$, dove *C* e *D* sono due numeri di *n* bit. La logica addizionale è costituita in questo caso da una porta XNOR; quando i due bit più significativi sono diversi è infatti possibile risolvere l'uscita indipendentemente dal valore logico assunto dagli altri ingressi, ed il registro *R*2 viene disabilitato. Assumendo che C_{n-1} e D_{n-1} abbiano entrambi probabilità di segnale pari a 0.5, la probabilità di predire correttamente il valore dell'uscita utilizzando solo i due bit più significativi è del 50% indipendentemente da *n*. Ciò significa che è possibile ottenere una riduzione di potenza pari al 50%, trascurando il contributo dato dalla logica di controllo.

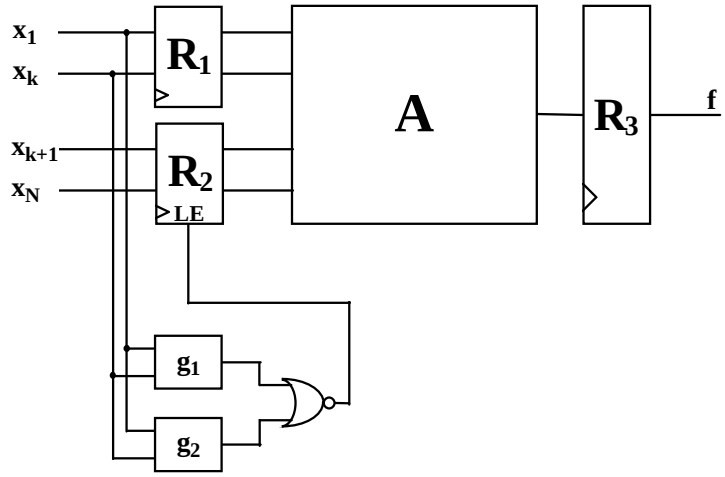


Figura 1.17{Fig_Precomp3}. Seconda architettura di "pre-computation".

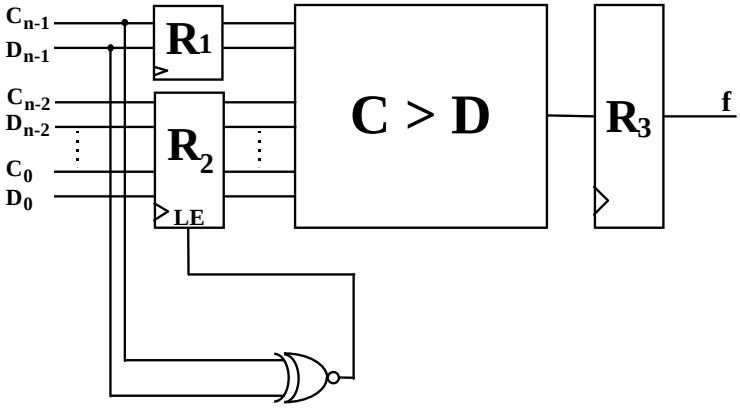


Figura 1.18{Fig_Precomp4}. Comparatore con "pre-computation".

Per determinare il sotto insieme {*x*1, ...*x*k} degli ingressi dei due predittori si valuta la probabilità della somma *g*1+*g*2 al variare di *k* e si sceglie la configurazione la cui probabilità è più vicina ad uno. Le due funzioni *g*1 e *g*2 si ricavano poi con le seguenti relazioni:

(1.23{Equ_a23})
$$g_1 = \forall_{x_{k+1},\ldots,x_N} f$$
$$g_2 = \forall_{x_{k+1},\ldots,x_N} \overline{f}$$

dove:

(1.24{Equ_a24}) $\forall_{x_{k+1},\ldots,x_N} f = \forall_{x_{k+1} \wedge x_{k+2} \wedge \ldots \wedge x_N} f$,

e $\forall_{x_i} f$ è il quantificatore universale di *f* rispetto alla variabile *xi* definito come:

(1.25{Equ_a25}) $\forall_{x_i} f = f(x_i = 1) \cdot f(x_i = 0)$.

Le architetture proposte possono essere facilmente estese anche ad altre strutture come, ad esempio, a reti sequenziali oppure a reti con uscite multiple. La riduzione in termini di potenza dissipata può arrivare anche al 60% con un impatto trascurabile sull'area.

Gated clock

Il principio alla base di questo metodo è quello di inibire selettivamente il segnale di *clock* dei registri quando la loro uscita non è significativa o deve rimanere invariata nel ciclo successivo. In altri termini il segnale di *clock* viene disabilitato quando la rete si trova in uno stato di attesa (*idle*), evitando così il riassegnamento.

Supponendo che la struttura del circuito sequenziale di partenza sia quella mostrata in figura 1.19a, la corrispondente versione con *clock* controllato è quella riportata in figura 1.19b. Il blocco combinatorio F_a, dipendente dagli ingressi primari e dallo stato presente, assume il valore uno in corrispondenza delle condizioni di *idle* del circuito. F_a è chiamata, solitamente, funzione di attivazione del segnale di *clock*. *L* è un *latch* introdotto per filtrare eventuali *glitch* che possono presentarsi all'uscita del blocco F_a. La funzione di attivazione viene quindi utilizzata, in AND con *CLK*, per comandare il *pin* di *clock* del registro di stato.

Uno dei problemi che si incontrano nell'utilizzare una struttura di questo genere è quello di individuare le condizioni di attesa in cui è possibile arrestare il *clock*. La soluzione proposta in [Beni94] si basa sulla conoscenza del diagramma delle transizioni di stato (STD) della macchina a stati finiti associata alla rete sincrona da ottimizzare; in particolare per rilevare automaticamente le condizioni di *idle* si sfruttano gli autoanelli presenti nello STD.

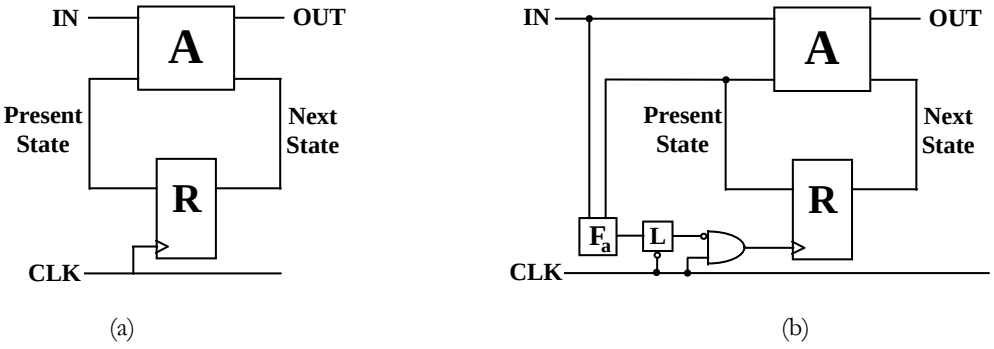


Figura 1.19{Fig_GatedC}. Circuito sequenziale (figura 1.19a) e versione con inibizione del clock (figura 1.19b).

Per una macchina di Moore un autoanello corrisponde sicuramente ad una attesa perché né l'uscita né lo stato presente commutano. La stessa cosa non vale invece per una macchina di Mealy in quanto non si può garantire che l'uscita non vari pur rimanendo nello stesso stato di partenza.

Molto semplicemente si può operare, in quest'ultimo caso, una trasformazione Mealy-Moore solo sugli stati dotati di autoanelli (altrimenti il numero di stati della macchina di Moore equivalente esploderebbe essendo pari a 2^O dove O è il numero di uscite primarie del circuito). Quando questi stati sono ancora in numero eccessivo, la trasformazione viene applicata solo agli autoanelli che sono attraversati con probabilità massima. Il numero degli stati viene così tenuto sotto controllo, tuttavia è necessario calcolare le probabilità di transizione di stato della FSM.

C a p i t o l o 4

LA SINTESI AD ALTO LIVELLO

INTRODUZIONE

Sintesi ad alto livello: è il passaggio da una specifica *a livello algoritmico* del *comportamento* di un sistema digitale ad una struttura RTL che implementi questo comportamento.

La specifica d'ingresso fornisce le trasposizioni (mappings) da sequenze d'ingresso a sequenze d'uscita; ingressi e uscite possono poi comunicare con l'ambiente esterno o con altri componenti a livello di sistema. La specifica dovrebbe imporre il minimo numero possibile di vincoli sulla struttura interna del sistema che si vuole progettare.

La struttura tipica di un sistema dedicato è costituita da:

- data path - insieme di registri (e altri dispositivi di memoria) e unità funzionali interconnessi mediante una rete realizzata con collegamenti diretti e multiplexer o con bus;
- unità di controllo (macchina a stati finiti).

Il prodotto della sintesi ad alto livello è costituito da:

- vista strutturale del data path
- specifica a livello logico (tabella degli stati) dell'unità di controllo.

La sintesi ottimale (rispetto a predefinite cifre di merito) consiste nell'identificazione della struttura che meglio soddisfa i vincoli (ad esempio: tempo di ciclo; area; potenza di alimentazione) pur minimizzando gli altri costi. Ad esempio, si richiedono la minimizzazione dell'area e il contemporaneo raggiungimento di una data frequenza di operazione.

Le cifre di merito temporali sono:

- ***latenza*** – l'intervallo di tempo (misurato in numero di cicli) che intercorre fra la presentazione dei dati e la disponibilità dei risultati;
- ***throughput*** (frequenza di computazione): indica la frequenza alla quale i successivi insiemi di risultati sono disponibili, nel caso di circuiti che devono ripetere la stessa computazione su successivi insiemi di dati. (Definizione alternativa: intervallo di tempo fra due successivi insiemi di risultati - reciproco della frequenza). È di particolare interesse per circuiti sincroni che realizzano una *sequenza di operazioni* in modo ***pipeline*** (i segmenti del circuito operano simultaneamente su diversi insiemi di dati).

Le diverse soluzioni di un problema di sintesi definiscono lo *spazio di progetto* del problema stesso, costituito da un insieme discreto di *punti di progetto* a ognuno dei quali sono associati valori di area e prestazioni.

Si possono definire funzioni di valutazione corrispondenti agli *obiettivi* fissati per il progetto. Lo *spazio di valutazione del progetto* ha per dimensioni tali obiettivi. L'*ottimizzazione di un circuito*: tende a ottimizzare tutti gli obiettivi.

Non si confonda *sintesi ad alto livello* con *sintesi RTL* (in questo caso, registri, unità funzionali e relativi trasferimenti sono già in buona misura specificati). All'altro estremo gerarchico, la *sintesi a livello di sistema* affronta il partizionamento di un algoritmo in processi, realizzabili come moduli software su dispositivi programmabili generici o come circuiti dedicati, capaci di operare in parallelo o in pipeline, etc.; si passa al dominio del cosiddetto *hardware/software codesign*.

Si verifica oggi una tendenza verso uno spostamento del processo di sintesi automatica verso livelli di astrazione sempre più alti, dovuta a varie ragioni:

- ***abbreviare il ciclo di progetto***: più automatizzato è il processo di progettazione, più breve è il tempo richiesto per completare un progetto e quindi per portarlo sul mercato. Inoltre, dato che una parte significativa del costo di un dispositivo sta nel suo progetto, automatizzare una parte rilevante del progetto significa ridurre in modo significativo i costi.
- ***diminuire la possibilità di errore***: se è possibile verificare la correttezza del processo di sintesi ci sono maggiori garanzie che il progetto finale corrisponda alle specifiche iniziali ⇒ si riducono gli errori e quindi il tempo per il "debugging".
- aumentare la ***possibilità di esplorare lo spazio del progetto:*** la sintesi consente di ottenere più progetti diversi a partire da una data specifica, in un tempo ragionevole. Il progettista può esplorare diversi bilanci fra costi, velocità, consumo, etc, oppure prendere un progetto esistente e produrne uno funzionalmente equivalente ma più veloce o meno costoso.
- aumentare la ***possibilità di documentare il processo di progettazione***: un sistema di sintesi automatica può seguire le scelte fatte e gli effetti che esse producono.
- consentire l'***accesso alla tecnologia VLSI per un'utenza più ampia***: diventa possibile anche per progettisti meno esperti produrre un chip che soddisfi specifiche assegnate.

LA DEFINIZIONE DEL SISTEMA DA SINTETIZZARE

Il sistema da progettare è solitamente descritto a livello algoritmico in un linguaggio di programmazione o in un linguaggio quale VHDL. Il linguaggio di specifica deve consentire di specificare gerarchia e task concorrenti; tali scomposizioni logiche consentono di comprendere meglio il comportamento del sistema ma sono raramente la soluzione migliore per una scomposizione hardware.

Il sistema di sintesi deve partizionare il progetto in procedure e task concorrenti secondo criteri di ottimalità del progetto hardware. La fase di pianificazione e partizionamento è una delle più difficili in un sistema di sintesi ad alto livello. Un problema particolarmente difficile è la scomposizione di un algoritmo in moduli concorrenti.

Il primo passo della sintesi consiste nella compilazione dal linguaggio formale a una rappresentazione interna. In genere si adottano rappresentazioni basate su grafi, che mostrino sia il *flusso dei dati* (Data Flow Graph, DFG) che il *flusso del controllo* (Control Flow Graph, CFG)

impliciti nella specifica. Ci sono molte varianti di DFG e CFG, e differenze nella quantità di informazione mantenuta dalla specifica: ad esempio, alcuni CFG non includono tutte le dipendenze di controllo nel programma ma solo quelle essenziali come i salti condizionati, alcuni DFG non usano gli assegnamenti di variabili definiti nella specifica per definire gli ordinamenti essenziali delle operazioni, etc.

Lo scopo di tutte le alternative è comunque quella di raggiungere una rappresentazione intermedia comune utilizzabili per le successive fasi della sintesi, e cioè:

- **Ottimizzazione iniziale** della rappresentazione intermedia. Queste trasformazioni ad alto livello includono eliminazione del "codice morto" (istruzioni mai eseguite), propagazione delle costanti, eliminazione delle sottoespressioni comuni, espansione in linea delle procedure e "srotolamento" (unrolling) dei cicli, oltre a trasformazioni locali più specifiche dello hardware.
- **Scheduling**: comporta l'assegnamento delle operazioni ai passi di controllo (passo di controllo: unità elementare di sequenziamento in un sistema sincrono, cioè ciclo di clock).
- **Allocazione**: comporta l'assegnamento di operazioni e valori a risorse hardware, quindi l'introduzione di unità funzionali, memorie, percorsi di comunicazione, e definizione del loro uso.

Scheduling e allocazione prefigurano una *scelta architetturale* per la struttura finale: tale scelta non deve influenzare la forma canonica di rappresentazione, ma è indispensabile per le fasi successive.

I MODELLI ARCHITETTURALI NELLA SINTESI

Nel passare dalle specifiche ad alto livello al progetto strutturale basato su componenti standard, si sceglie uno stile di progetto e si definisce poi una architettura obiettivo.

- stile di progetto: caratteristiche qualitative del progetto (es., per il progetto di un microprocessore: interruzione con livelli di priorità, progetto basato su un bus, I/O di tipo seriale, etc.)
- architettura obiettivo: definisce il progetto in termine di particolari unità, dei loro parametri e dei collegamenti che le connettono (nell'esempio, l'architettura comprende numero di registri, numero di bus, numero dei bit di stato, etc.).

Programmi e algoritmi di sintesi ipotizzano una (o alcune) architettura obiettivo: un'architettura che traspone in modo diretto la descrizione comportamentale su uno schema strutturale è di norma inutilmente costosa. Il miglioramento del rapporto costo/prestazioni richiede procedure di sintesi più complesse.

Es.: si considerino due architetture di "datapath" per un semplice microprocessore:

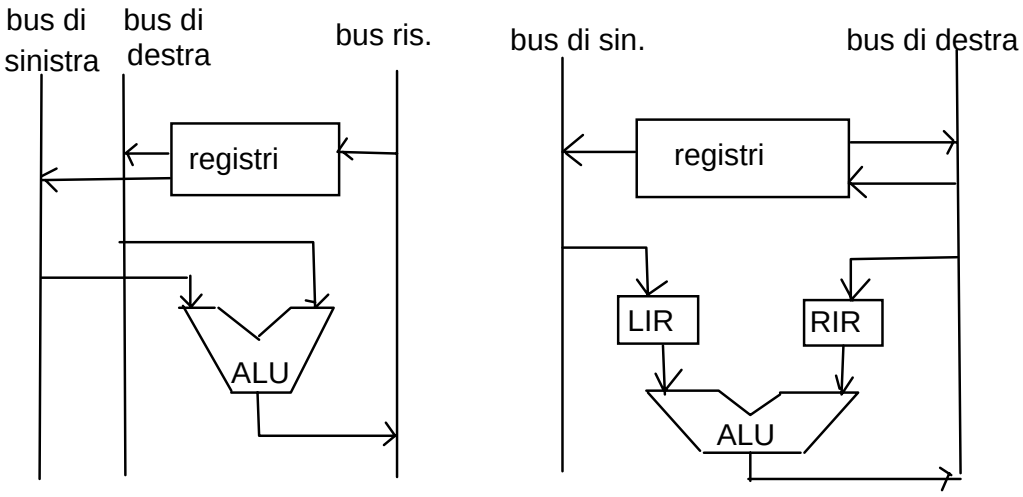


Figura 1 a) b)

Si consideri dapprima l'architettura in figura 1.a. È possibile richiamare simultaneamente due operandi dal banco di registri, comandare un'operazione su di essi da parte dell'ALU e riportare il risultato nel banco di registri in un unico ciclo di clock (nell'ipotesi che i registri siano di tipo master-slave).

Si supponga che le variabili *a, b, c, x, y* siano memorizzate in registri arbitrari e che il ciclo di clock sia di 100 nsec. Le due operazioni binarie:

$x \Leftarrow a+b$ (100 nsec)

$y \Leftarrow c-x$ (100 nsec)

possono essere eseguite in due cicli di clock (200 nsec).

Il datapath di figura 1.a è poco realistico: un bus occupa area elevata, provoca carichi capacitivi (quindi diminuisce la velocità) - tre bus non darebbero un uso efficiente del silicio. Inoltre, registri e ALU sono usati solo in una parte del ciclo e in modo mutuamente esclusivo: quando si propagano segnali attraverso l'ALU, non se ne propagano attraverso i registri, e viceversa.

Si modifica l'architettura inserendo i due registri RIR e LIR a monte dell'ALU, dimezzando il ciclo di clock e aumentando l'uso di registri e ALU in ogni ciclo (fig. b). Con questo modello, le precedenti operazioni diventano:

LIR$\Leftarrow a$; RIR$\Leftarrow b$; (50 nsec)

x, RIR$\Leftarrow$LIR+RIR; LIR$\Leftarrow c$; (50 nsec)

$y \Leftarrow$LIR-RIR; (50 nsec)

e possono essere eseguite in tre cicli (150 nsec) con un miglioramento delle prestazioni del 25%. Si noti però che se non ci fosse stata dipendenza fra le due operazioni, non ci sarebbe stato miglioramento: le operazioni $x \Leftarrow a+b$ e $y \Leftarrow c-d$ richiederebbero due cicli sulla prima architettura e quattro (totale 200 nsec) sulla seconda.

Il modello di figura 1.a richiede un algoritmo molto semplice per assegnare le operazioni ai vari cicli di clock (operazione detta scheduling); il modello in figura 1.b è più realistico ma richiede

- una procedura di sintesi che identifichi le relazioni di dipendenza fra le operazioni;
- un algoritmo di scheduling che tenti un miglioramento delle prestazioni riordinando la sequenza di esecuzione delle operazioni sulla base delle dipendenze.

I modelli architetturali ai vari livelli di astrazione

I dispositivi VLSI constano di unità funzionali e unità di memoria: nelle "librerie" utilizzate dai sistemi di CAD per la sintesi compaiono varie soluzioni per le principali unità delle due classi, nella tecnologia che caratterizza la libreria.

Le **unità di memoria fondamentali** sono:

- **registri:** (possono essere letti e scritti individualmente, con un parallelismo legato solo alla rete di interconnessione),
- **banchi di registri:** (register files), dotati anche del meccanismo di indirizzamento e di accesso ai dati: possono avere un numero predefinito di porte *distinte* di lettura e scrittura, consentono operazioni di lettura e/o scrittura simultanee alle diverse porte (se si tratta di registri master-slave, lettura e scrittura simultanee possono riguardare anche lo stesso registro);
- **memorie RAM e ROM;** normalmente hanno una sola porta di accesso (più raramente due); non è possibile accedere simultaneamente da due porte alla stessa parola, se non in lettura.

Le **unità funzionali fondamentali** possono essere suddivise in unità *nonsliceable* - che non possono essere partizionate in blocchi relativi a un singolo bit e collegati l'uno all'altro secondo uno schema ripetitivo di interconnessione - e unità *sliceable*.

- *Unità nonsliceable*: sono, ad esempio, codificatori, decodificatori, generatori del riporto in addizionatori ad anticipazione di riporto.
- *Unità sliceable*: es., addizionatori, generatori di parità, confrontatori.

(ad esempio, un sommatore può essere realizzato componendo unità della prima e della seconda classe).

Nella sintesi, le *unità funzionali* sono viste come blocchi di logica combinatoria, descrivibili mediante funzioni logiche degli ingressi (non necessariamente in corrispondenza uno-a-uno con l'implementazione) o mediante la tabella delle verità.

Le macchine a stati finiti (FSM)

Il concetto di *macchina a stati finiti* (finite state machine – FSM) è il più conosciuto modello di progetto impiegato nell'ingegneria per descrivere reti sequenziali e può essere usato per rappresentare diversi stili architetturali. Consiste di un insieme di stati, di un insieme di transizioni e di un insieme di azioni associate agli stati e alle transizioni. In maniera formale si hanno le seguenti definizioni.

Definizione Una *macchina a stati finiti*, M, è definita come una quintupla:

$$M = (I, U, S, S^0, R)$$

dove $I \subseteq B^n$ rappresenta l'alfabeto d'ingresso, $U \subseteq B^m$ l'alfabeto d'uscita, $S \subseteq B^k$ l'insieme degli stati, S^0 è lo stato iniziale ed R la relazione globale.

Definizione La *relazione globale* é la relazione

$$R \subseteq S \times I \times S \times U \rightarrow \{0,1\}$$

tale che $R(i,u,s,t) = 1$ se e solo se, applicando l'ingresso $i = (i_1,i_2,..,i_n) \in I$, la macchina a stati M passa dallo stato presente $s = s_1,s_2,..,s_k \in S$ allo stato prossimo $t = t_1,t_2,..,t_k \in S$ generando all'uscita il valore $u = u_1,u_2,..,u_m \in U$.

Definizione Una macchina a stati finiti M può essere rappresentata dal *grafo delle transizioni di stato* i cui nodi sono formati dagli elementi $s \in S$ e i cui archi sono etichettati con le coppie $(i,u) \in I \times U$.

Definizione Data la relazione globale della macchina a stati finiti, la *relazione di transizione di stato* vale:

$$\Delta(i,s,t) = \exists u \; R(i,u,s,t)$$

e fornisce il legame fra lo stato corrente e lo stato prossimo fissato un determinato valore di ingresso.

Definizione Data la relazione globale della macchina a stati finiti, la *relazione d'uscita* vale:

$$\Lambda(i,u,s) = \exists t \; R(i,u,s,t)$$

e fornisce il valore d'uscita di M fissato lo stato iniziale e il valore d'ingresso.

Le funzioni di stato prossimo e di uscita si indicano rispettivamente con δ, λ.

Le macchine a stati finiti in cui la relazione di uscita Λ dipende solo dallo stato presente sono dette *macchine di Moore* (FSM state-based), mentre quelle in cui lo stato dipende anche dal valore degli ingressi sono dette *macchine di Mealy* (FSM transition-based). E' sempre possibile passare da una rappresentazione di Mealy ad una di Moore e viceversa. Un caso particolare è rappresentato dalle FSM autonome dove l'alfabeto d'ingresso I è vuoto. Esempi tipici sono contatori e divisori di frequenza.

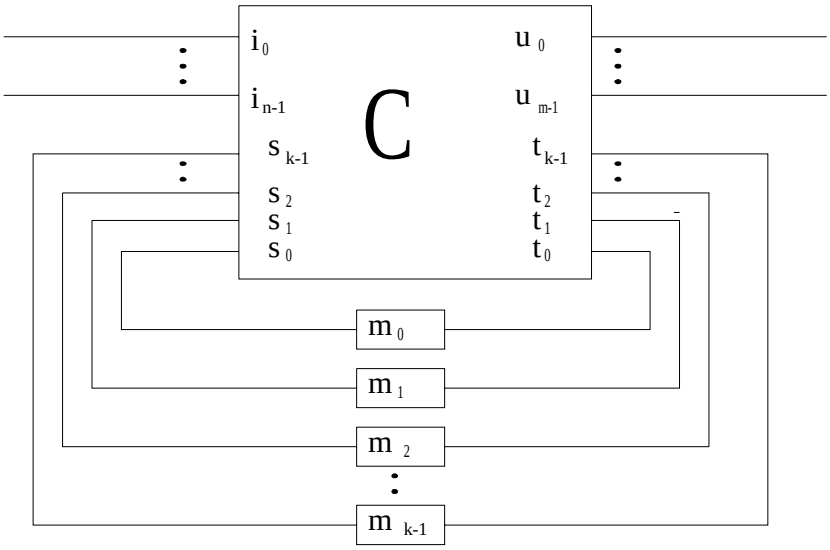


Figura 20{Fig_cap2_fig1}. Schema tipico di un circuito sequenziale (modello di Huffman); *C* è la rete combinatoria, $i_0,..,i_{n-1}$ sono le variabili di ingresso, $u_0,..,u_{m-1}$ sono le variabili di uscita, $s_0,..,s_{k-1}$ sono le variabili di stato presente, $t_0,..,t_{k-1}$ sono le variabili di stato prossimo ed $m_0,..,m_{k-1}$ sono gli elementi di memoria.

Alla descrizione formale data sino ad ora di macchina a stati corrispondono nella realtà vari tipi macchine sequenziali. Le più diffuse sono le macchine sequenziali *sincrone* dove esiste una base dei tempi, un segnale impulsivo a frequenza fissa detto *clock* che non costituisce ingresso di informazione, ma che scandisce gli eventi a cui la rete è soggetta. All'opposto esistono le macchine *asincrone* in cui non esiste questa base dei tempi e che identificano gli eventi attraverso il riconoscimento di impulsi applicati agli ingressi (*macchine in modo impulsivo*) oppure attraverso il riconoscimento della transizione del livello di segnale su una linea (*macchine a livelli*). La realizzazione logica di una macchina a stati finiti memorizza gli stati negli elementi di memoria (flip-flop) mentre le funzioni logiche che descrivono la funzione di stato e la funzione di uscita sono implementate da una rete combinatoria. La struttura canonica di un circuito sequenziale sincrono, denominata schema di Huffman, è riportata in figura 20.

Il modello FSM è gestibile dal progettista fino a qualche centinaia di stati; interfacce di I/O, controllori di bus etc. possono avere diverse migliaia di stati. Perciò per rendere il modello comprensibile, si introducono registri, banchi di registri e memorie che possono ospitare variabili i cui diversi valori corrispondono a diversi stati giungendo così al concetto di macchina a stati più datapath.

Le FSM con Datapath come modello architetturale

Il modello di macchina a stati finiti prima presentato è adatto per descrivere reti sequenziali con poche centinaia di stati. Un numero di stati maggiore diventa non gestibile da un qualsiasi progettista. Inoltre, anche componenti non tanto complessi come per esempio le interfacce di ingresso uscita o i circuiti di controllo del bus possono presentare molti stati se si considerano tutti gli elementi di memorizzazione. Allo scopo di rendere utilizzabile il modello basato sulla macchina a stati finiti per descrivere progetti complessi, in [15] si è introdotto un insieme di variabili intere o in virgola mobile che possono essere memorizzate nei registri, nei register-file

e nella memoria. In questo modo ogni variabile sostituisce migliaia di stati, per esempio, una variabile intera codificata con 16 bit rappresenta $2^{16} = 65536$ stati diversi.

Si indichi con *Var* questo insieme di variabili, con $Exp = \{ \Delta(x, y, z, \ldots) \mid x, y, z, \ldots \in Var\}$ un insieme di funzioni e con $Asg = \{ X = e \mid X \in Var, e \in Exp \}$ un insieme di assegnamenti. Infine si definisca l'insieme di *variabili di stato* come la relazione fra le funzioni dell'insieme *Exp*:

$$Stat = \{ (a, b) \mid a, b \in Exp \}$$

Quindi una macchina a stati finiti con unità di elaborazione (FSMD) risulta essere definita dalla quintupla:

$$<S, I \cup B, O \cup A, \Delta, \Lambda>$$

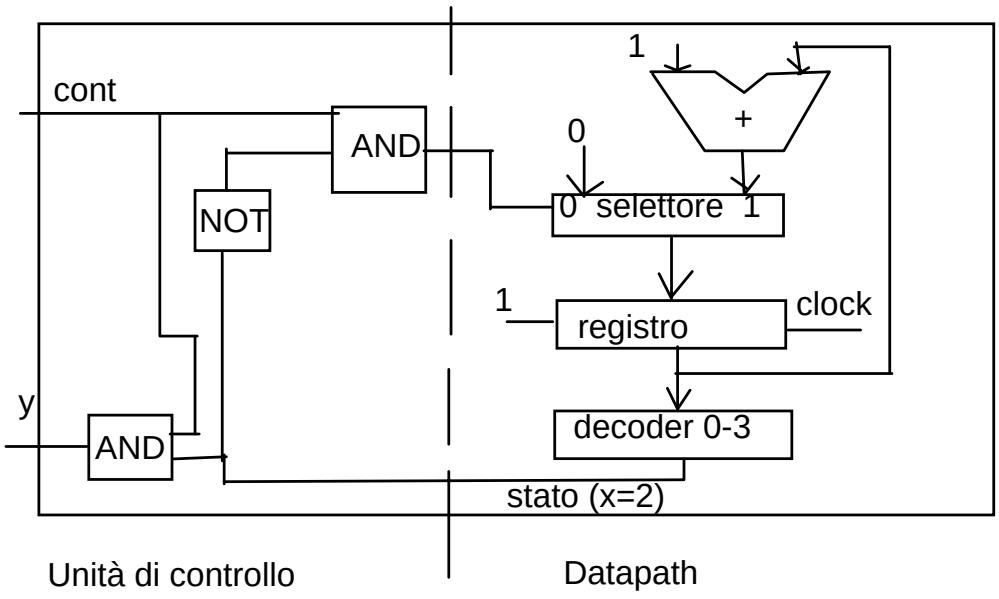
dove S, Δ, e Λ rappresentano gli stessi insiemi definiti per una FSM e dove l'insieme dei valori d'ingresso è esteso in modo da includere alcune delle variabili di stato, $B \subseteq Stat$, e l'insieme delle uscite include alcuni assegnamenti, $A \subseteq Asg$.

La FSMD è costituita da una *unità di controllo* (FSM) che scambia informazioni di controllo con un *datapath*, cui è delegata la realizzazione delle funzioni e che include unità funzionali e memorie.

Es.: Si modelli un divisore di frequenza modulo 3 come una FSMD. Il divisore ha un ingresso di clock, un ingresso di reset *cont* e un'uscita *y*.

***Primo modello*:** la FSM controllante ha un solo stato (= rete combinatoria). Nel datapath si introduce una variabile *x* (registrata in un registro) di due bit: se $x \neq 2$, la si incrementa di 1, se $x=2$ si pone $x=0$. Si danno in tabella le funzioni stato prossimo e uscita, e in figura lo schema della FSMD.

Stato pres.	Ingresso	Stato pross.	uscita
s0	(cont=1)AND(x≠2) (cont=1)AND(x=2) cont=0	s0	x=x+1,Y=0 x=0,Y=1 x=0,Y=0



Figura

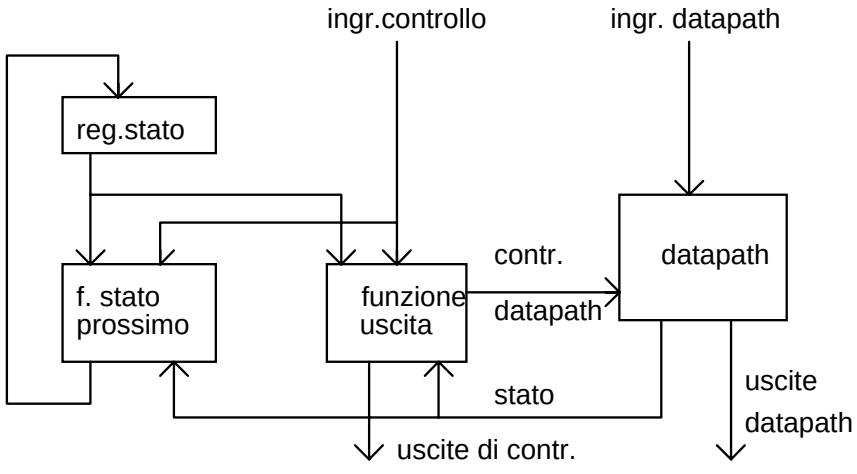
Il datapath è costituito da un registro che memorizza *x*, un MUX che trasferisce *1* o *x+1* nel registro, un addizionatore che incrementa il valore del registro.

Quando cont=1, nel registro si carica o 0 (se la *x* vale 2, - l'informazione viene prelevata all'uscita del decoder) oppure *x*+1.

Quando cont=0, il valore del registro viene forzato a 0.

È possibile modellare lo stesso divisore di frequenza mediante una FSMD con tre stati, usando uno stato diverso per ogni distinta combinazione di valori d'uscita e contenuto dei registri. In pratica, si sposta il contatore dal datapath al controllore.

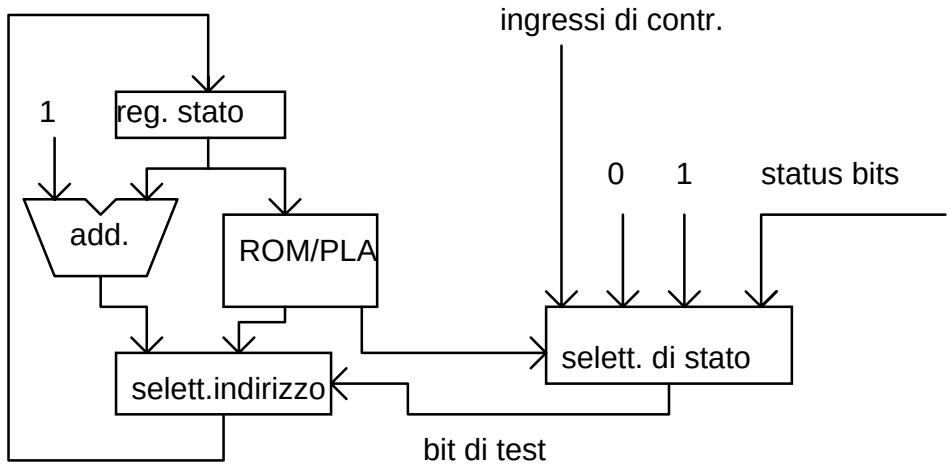
Le FSMD sono usate per descrivere sistemi digitali a livello RTL. Lo schema a blocchi generico è il seguente:



Figura

costituito da una *unità di controllo* e da un *datapath*. Gli ingressi all'unità di controllo si dividono in *ingressi di controllo* e *ingressi di stato*: le uscite si dividono in *uscite di controllo* e *segnali di controllo per il datapath*. Ingressi e uscite del datapath sono tipicamente parole, mentre ingressi e uscite di controllo sono spesso bit singoli.

La funzione stato prossimo viene spesso realizzata tenendo conto che lo stato prossimo può essere ottenuto o incrementando quello presente o imponendo una variazione nel caso di un "salto": la scelta fra le due alternative dipende sia dagli ingressi di controllo che da quelli di stato. Una soluzione adottata nel caso di molti microcontrollori o microprocessori è:



Figura

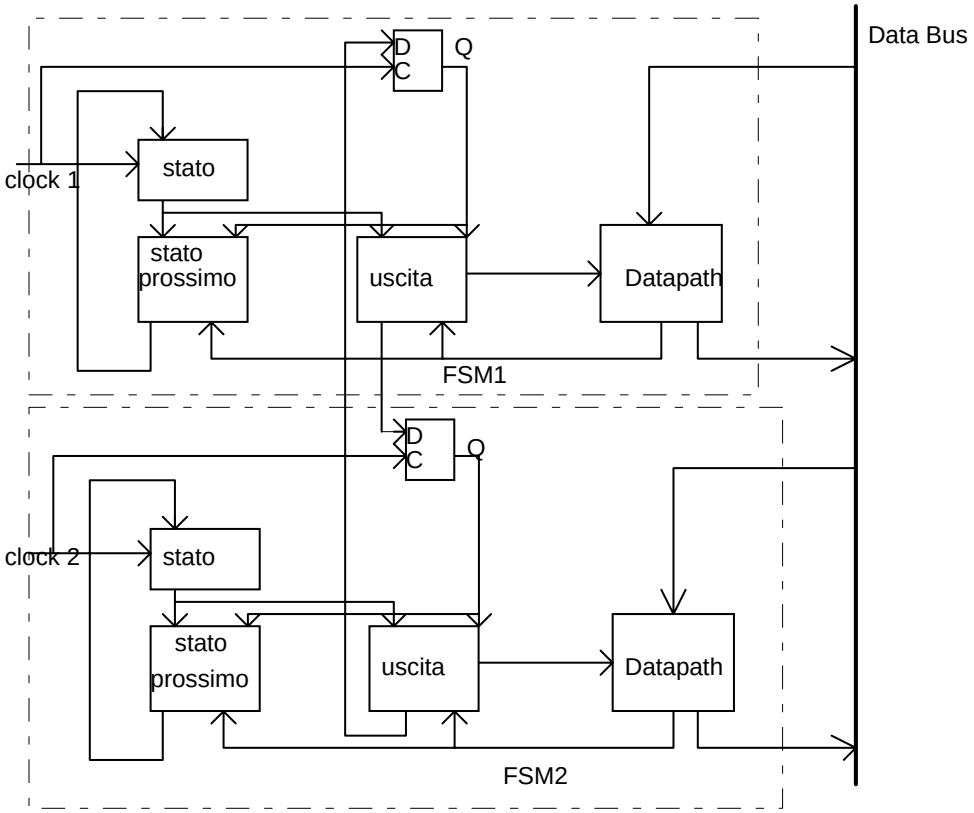
Il bit di test (che indica se lo stato prossimo si determina per incremento o come "salto") è selezionato dal selettore di stato fra ingressi di controllo (es. interruzioni esterne) e bit di stato (es.overflow, bit di segno, etc.)

A livello di astrazione più alto, la FSMD può essere vista come un processo che "consuma" ingressi e "produce" uscite. Un sistema può essere descritto mediante un insieme di processi comunicanti, descritti ognuno mediante costrutti standard di linguaggi di programmazione.

La descrizione comportamentale presuppone l'esistenza di elementi di memoria solo per tutte le variabili globali e locali e ipotizza che operazioni, letture e scritture di valori, trasferimenti non richiedano tempo. Il modello non include cioè i concetti di ritardo, intervallo di tempo, stato: il concetto di tempo è ridotto all'ordinamento per l'esecuzione delle istruzioni.

Un sistema descritto come insieme di processi, procedure e componenti viene partizionato in sottodescrizioni, corrispondenti a componenti di sistema ognuno dei quali può essere realizzato con una o più FSMD comunicanti. La comunicazione è fra unità di controllo o fra datapath o fra ambedue. I segnali di controllo raramente si usano come ingressi ai datapath di altri processi, e viceversa i valori dei datapath raramente si usano come segnali di controllo di altre unità di controllo.

Il numero di segnali e le relazioni temporali fra i segnali durante la comunicazione costituiscono un protocollo; se tutti i processi sono comandati da un unico clock, il sistema è sincrono; se le frequenze di clock sono diverse, le comunicazioni si dicono asincrone. Si consideri il seguente semplice esempio di due FSMDE comunicanti:



Figura

In ambedue i tipi di comunicazioni, il protocollo più usato è quello detto di *handshaking di tipo request-acknowledge*:

La prima FSMD interrompe la seconda portando a 1 il segnale di request; resta in stato di request fino a quando la seconda FSMD pone a 1 il segnale di acknowledge, e a questo punto lascia lo stato di request e porta a 0 il corrispondente segnale. La seconda FSMD passa nello stato di acknowledge e vi resta fino a quando il segnale di request non viene rimosso - allora lascia lo stato di acknowledge. La sequenza request-acknowledge può essere usata anche per controllare i trasferimenti sul bus dati.

La sintesi include i seguenti passi:

1) Compilazione (che traduce la descrizione in un linguaggio ad alto livello in una forma standard, quale i grafi di Data Flow e Control Flow che si useranno qui);
2) Partizionamento in gruppi locali (allocati su diversi chip o, oggi più facilmente, in diversi sottocircuiti) che possono a loro volta essere partizionati in più sottosistemi sintetizzati mediante architetture FSMD;
3) Sintesi delle interfacce, mediante cui si determinano canali e protocolli di comunicazione e si genera l'hardware che li esegue;
4) Scheduling, che divide il Data and Control Flow Graph in stati, o "passi di controllo";
5) sintesi delle FSMD.

Se il tempo richiesto per un'operazione da parte di un'unità funzionale richiede *più di un ciclo di clock*, l'unità di controllo deve tenerne conto, in modo che il risultato venga "atteso" per tutti i

cicli di clock necessari per produrlo. L'ovvia conseguenza di un tale approccio è la sotto-utilizzazione del sistema: l'intera unità è bloccata per un tempo lungo, anche se solo una sua piccola parte è in un qualsiasi ciclo attiva.

Una soluzione che aumenta l'utilizzo delle unità funzionali è l'uso del ***pipelining***.

- l'unità è divisa in stadi
- i risultati parziali di ogni stadio vengono registrati in un latch;

il ciclo di clock può essere commisurato al tempo richiesto dallo stadio più lento.

Questa tecnica consente di operare simultaneamente su diverse coppie di operandi, ognuna delle quali viene elaborata parzialmente da un diverso stadio dell'unità "pipelined". Il *ritardo totale* (latenza) può aumentare leggermente a causa del tempo necessario per registrare i risultati parziali nei latch; il *throughput totale* su un flusso di operandi indipendenti (quindi la frequenza a cui si possono caricare nuove coppie di operandi) è proporzionale al numero di stadi.

LA RAPPRESENTAZIONE DEL PROGETTO

Data una descrizione comportamentale in un HDL, occorre una rappresentazione intermedia "canonica" che faciliti un mapping efficiente dalla descrizione d'ingresso a diverse architetture obiettivo e consenta l'uso di diversi strumenti di sintesi. La *rappresentazione intermedia canonica* deve *conservare il comportamento* della specifica iniziale e *consentire l'aggiunta dei risultati della sintesi* mediante raffinamenti, bindings, ottimizzazioni e mappings vari.

Comportamento d'ingresso, struttura sintetizzata e controllo sintetizzato rappresentano oggetti in diversi domini del progetto (tipicamente: data path e controllo), oltre che a diversi livelli: occorre correlare questi oggetti per poter effettuare simulazione e messa a punto di tipo multi-livello. Una rappresentazione intermedia ideale deve:

1) fungere da archivio di tutta l'informazione di progetto, nel corso della sintesi, inclusi comportamento originale, vincoli di progetto, strutture sintetizzate, binding fra diversi oggetti;
2) fornire una visione uniforme della rappresentazione, su insiemi di strumenti e di utenti diversi;
3) essere indipendente dall'HDL iniziale;
4) supportare diversi stili architetturali per la realizzazione finale.

In realtà, non è possibile soddisfare tutti questi requisiti (in parte contraddittori: ad esempio, la descrizione comportamentale varia con le primitive e i livelli di astrazione che caratterizzano un HDL).

Il primo passo è la **compilazione del comportamento (ingresso) in una rappresentazione intermedia basata su grafi**. Tali grafi vengono poi utilizzati per le operazioni di sintesi ad alto livello.

Se si parte da una descrizione di tipo comportamentale (es.: processi VHDL) né la descrizione VHDL iniziale, né il CDFG corrispondente indicano come il progetto dovrà essere realizzato in hardware: le variabili non sono legate a elementi di memoria; le operazioni non sono legate a unità funzionali. Inoltre, non vengono specificati la sequenziazione degli stati o i segnali di controllo che devono attivare i componenti del datapath in ogni stato.

La fase di compilazione HDL

Nella rappresentazione mediante CFDG, i costrutti per il flusso di controllo del VHDL vengono rappresentati mediante nodi del flusso di controllo, mentre i blocchi di istruzioni di assegnamento fra due costrutti di controllo vengono rappresentati mediante grafi di flusso dei dati. Mentre il grafo di flusso del controllo (CFG) è connesso e può contenere dei cicli, il grafo di flusso dei dati può essere non connesso (in genere non lo è) e non presenta cicli.

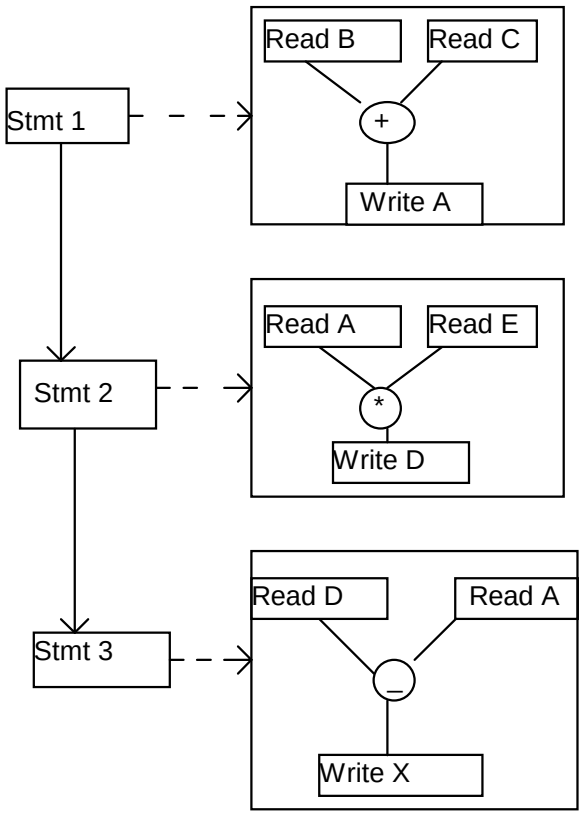
Si consideri una semplice sequenza di assegnamenti:

A:=B+C

D:=A*E

X:=D-A

La prima realizzazione del DFG deriva semplicemente dagli *alberi di parse o alberi sintattici* (il CFG è banale - costituito da tanti alberi quante sono le operazioni binarie e gli assegnamenti estratti dal processo; il DFG identifica le successive istruzioni del processo). Il parser del VHDL genera dei parse trees con una trasformazione uno-a-uno dalle istruzioni:



Figura

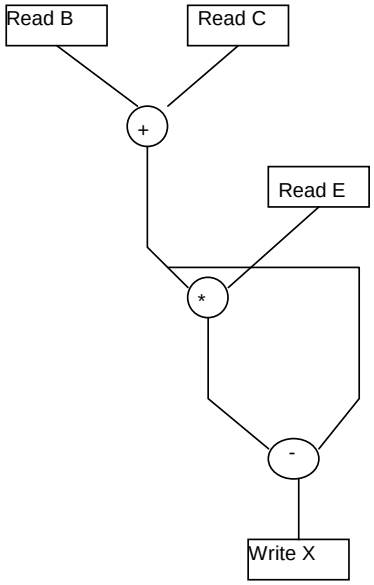
L'ordinamento di esecuzione degli alberi di parse viene interpretato sulla base dello stile di esecuzione adottato in VHDL; ad esempio, se lo stile fosse data-flow, con una semantica che richiede l'esecuzione in parallelo delle istruzioni, gli alberi di parse verrebbero analizzati per garantire che tutte le espressioni che compaiono a destra nelle istruzioni di assegnamento vengano valutate simultaneamente, prima di assegnare i valori ottenuti alle variabili sulla sinistra delle istruzioni. Gli alberi vengono mantenuti intatti, salvo la fusione delle variabili

comuni sul lato destro delle istruzioni. Si prevede una preelaborazione che riduca tutte le istruzioni aritmetiche a *costrutti aritmetici semplici* (operazioni con uno o due operandi).

Se i costrutti linguistici sono di tipo sequenziale (stile behavioral in VHDL), si compie un'analisi del flusso di dati sugli alberi per identificare il potenziale parallelismo fra le istruzioni sequenziali. Nell'esempio, si vede che A è definita nella prima istruzione e usata nella seconda e nella terza - quindi c'è una dipendenza di dati "in avanti" per A da Stmt1 a Stmt2 e Stmt3; analogamente, c'è una dipendenza in avanti su D fra seconda e terza istruzione.

Il CFG non mette in rilievo il potenziale *parallelismo* fra istruzioni, che invece è identificabile in base alle *dipendenze di dati*.

L'analisi del flusso di dati viene completata fondendo tutti gli alberi di parse in un unico grafo di flusso dei dati, in cui si mantengono le dipendenze di dati e da cui si estrae il parallelismo.



Figura

Le varie alternative di CDFG esistenti si differenziano essenzialmente per il modo di rappresentare i trasferimenti dati entro il grafo di flusso dei dati e di rappresentare i costrutti di controllo. Si vedrà ora in modo intuitivo la rappresentazione del controllo, per analizzare poi alcuni schemi alternativi.

La rappresentazione del flusso di controllo

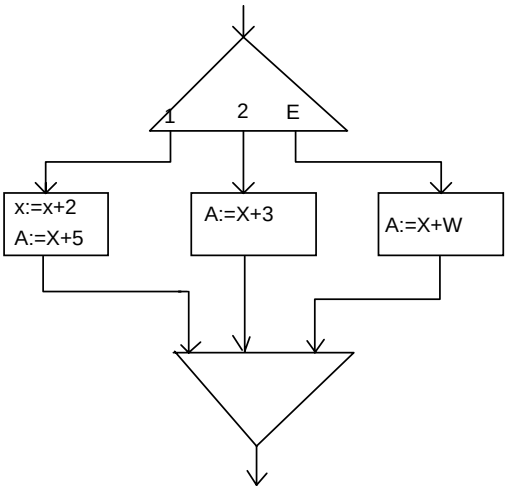
Sono possibile diverse alternative per catturare il flusso del controllo. Si esamini il seguente segmento di programma in VHDL:

```
case C is
    when 1⇒
        X:=X+2; A:=X+5;
    when 2⇒
        A:=X+3;
    when others ⇒
```

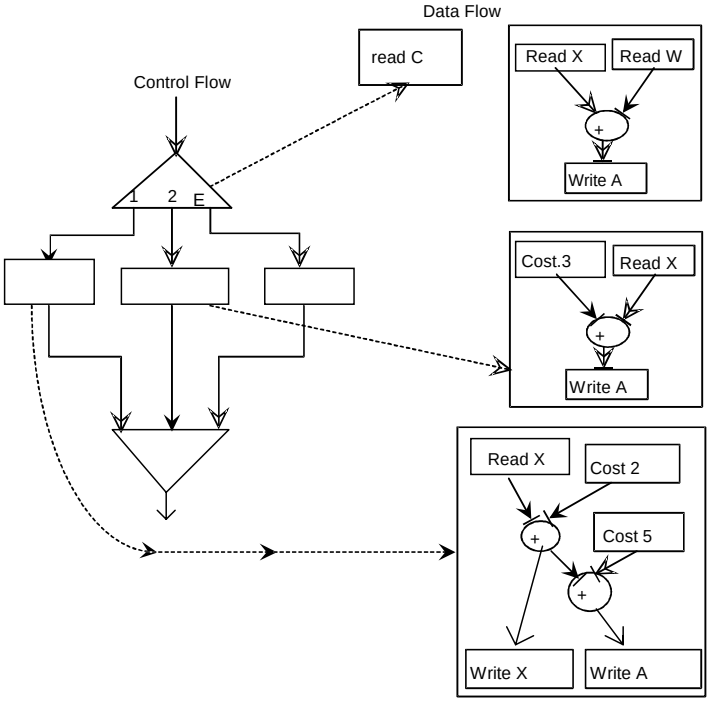
A:=X+W;

end case;

Primo schema: i costrutti di controllo vengono tradotti ("mapped") mediante nodi di controllo di flusso che mantengono sequenziamento e flusso di controllo espliciti come espressi nel programma:



Figura



Figura

Il grafo di flusso del controllo è costituito da nodi di diramazione condizionata (triangoli), nodi di join condizionato (triangoli capovolti) e blocchi di istruzioni di assegnamento (rettangoli). A ogni nodo può essere associato un blocco di flusso dei dati che ne descrive l'attività operativa.

Nel grafo disegnato, ogni percorso del *case* mostra in modo esplicito la mutua esclusione, e gli assegnamenti entro ogni valutazione condizionata, trasposti su blocchi di data flow.

La rappresentazione, molto prossima a quella iniziale, è facile da mettere a punto e collegare alla descrizione di partenza.

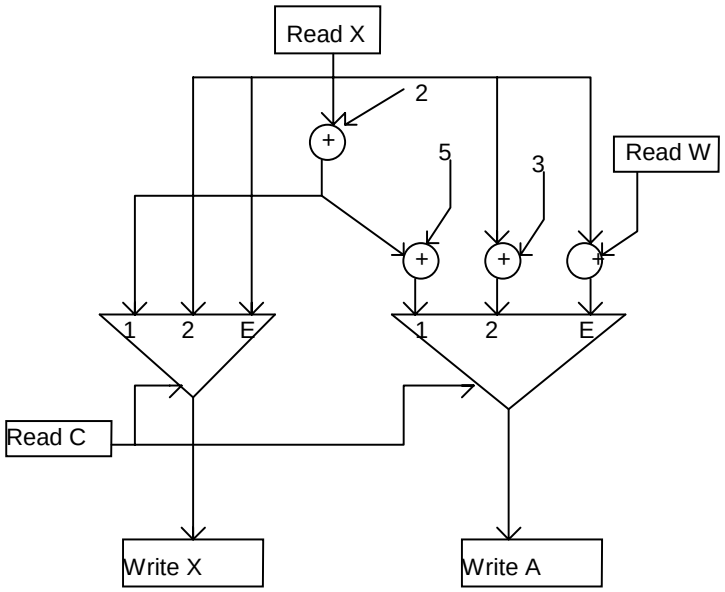
Secondo schema di rappresentazione: si traspongono i costrutti di controllo entro il grafo di flusso dei dati, valutando in parallelo tutti i rami delle diramazioni condizionate e scegliendo i valori corretti per gli assegnamenti dopo che tutti i rami sono stati eseguiti. Questo significa

- calcolare tutti i possibili valori per ogni variabile sul lato sinistro di un'espressione;
- scegliere il valore opportuno in base al valore della variabile di condizione.

Si usano:

- cerchi per indicare le operazioni;
- archi per indicare il flusso dei dati;
- rettangoli per indicare lettura e scrittura dei dati;
- triangoli capovolti per indicare la scelta dei dati sulla base del valore di una linea di controllo.

Si veda la nuova rappresentazione per lo stesso esempio di prima: i triangoli che indicano la scelta sul flusso dei dati scelgono il valore opportuno per le variabili X e A, che appaiono alla sinistra in almeno un'istruzione di assegnamento.



Figura

Anche la rappresentazione basata sul flusso di dati mostra esplicitamente il parallelismo dovuto alla mutua esclusione di diversi percorsi condizionali: rende però disponibile per ottimizzazioni ed elaborazioni una parte di grafo maggiore della precedente.

Lo svantaggio è che - quando si hanno più cicli annidati l'uno nell'altro - occorre generare tutti i corrispondenti livelli di selettori: questo può portare a grafi di flusso di dati troppo ingombranti e difficili da trattare.

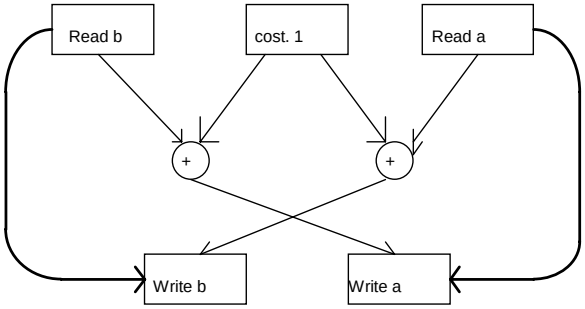
Si può anche usare una rappresentazione ibrida, in cui il flusso di controllo detta il sequenziamento dei nodi di flusso dei dati.

La rappresentazione del sequenziamento e della temporizzazione

La rappresentazione intermedia deve mantenere anche le relazioni di ordinamento specificate (in modo implicito o esplicito) nella descrizione d'ingresso. L'ordinamento può essere mostrato mediante archi fra i nodi di un grafo di flusso (gli archi di precedenza possono essere usati anche per forzare un ordinamento negli accessi alle matrici, quando i valori degli indici non sono costanti). Si consideri il seguente segmento scritto in VHDL concorrente:

```
b<=a+1
a<=b+1
```

Le istruzioni vengono eseguite in parallelo: occorre quindi garantire che le istruzioni di lettura precedano quelle di scrittura per i due segnali a e b. Gli archi più spessi in figura indicano le relazioni di precedenza:

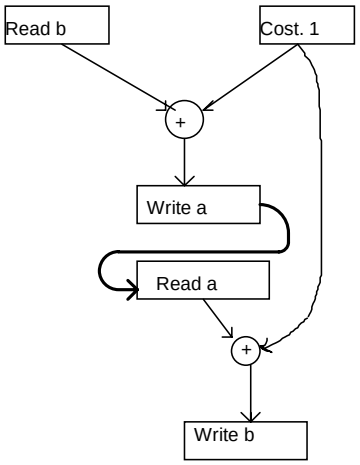


Figura

Nei diversi schemi di rappresentazione gli accessi alle variabili mediante letture e scritture vengono presentati in modo diverso. In alcuni casi si rendono espliciti dei nodi di accesso alle variabili; es., in un blocco che rappresenta un comportamento sequenziale, occorre che il valore di una variabile venga letto prima di definire un nuovo valore (e quindi fare una scrittura); viceversa, non si può leggere una variabile prima che questa sia stata definita (scritta). Per forzare l'ordinamento si possono utilizzare gli archi di precedenza: il segmento VHDL sequenziale

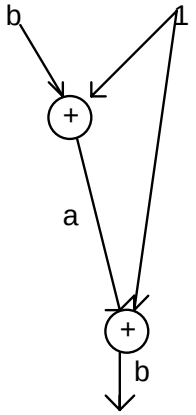
```
a:=b+1
b:=a+1
```

può essere rappresentato come segue:



Figura

Altri schemi basati su grafi di flusso rappresentano gli accessi a variabili in modo implicito, come tracce dei dati, usando archi nel grafo di flusso dei dati:

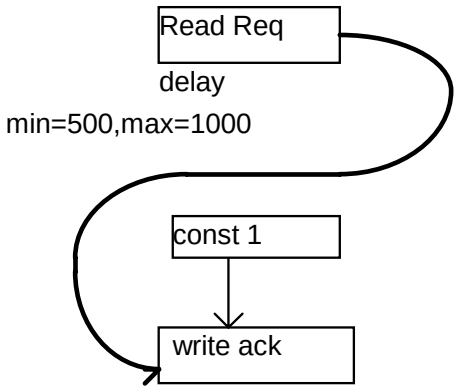


Figura

La prima soluzione (esplicita) rende più facile lo "unit binding"; la seconda (implicita) rende molto più facili le ottimizzazioni sul grafo di flusso dei dati.

Rappresentazione della temporizzazione: fornisce vincoli per scheduling, scelta delle unità e binding, oltre che per le prestazioni della tecnologia di realizzazione (es.: periodo di clock). Ci sono diverse alternative di rappresentazione, legate allo schema usato (e in particolare all'uso di Data-flow piuttosto che di Control-Flow graph).

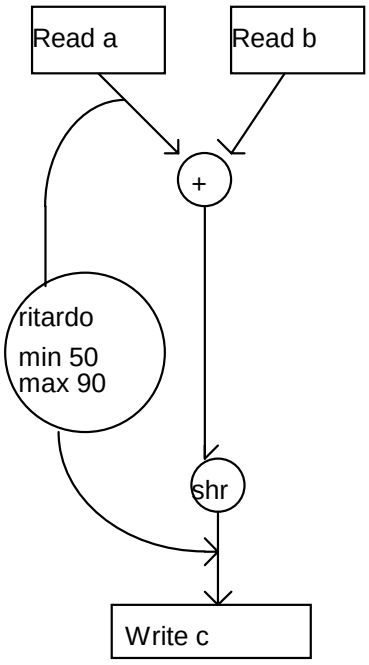
A livello di data-flow graph si può o annotare l'informazione temporale sugli archi di precedenza fra due nodi del grafo, o creare un nodo di temporizzazione fra due archi del grafo. La prima soluzione è utile per rappresentare i vincoli di temporizzazione minima, massima e nominale fra l'esecuzione di due operazioni (cioè i nodi) - particolarmente adeguata per rappresentare protocolli. Es:



Figura

Nel secondo schema, si usano *nodi espliciti di temporizzazione* fra archi del DFG, descrivendo i ritardi di percorso punto-a-punto entro il grafo, per fornire vincoli sui tempi minimi e massimi d'esecuzione dei singoli operatori oltre che per gruppi di operatori lungo il percorso di flusso dei dati dalla sorgente alla destinazione del nodo di temporizzazione.

Es.: nodo di temporizzazione fra ingresso di sinistra dell'operatore di addizione e uscita dell'operazione di scorrimento, che vincola l'esecuzione delle due operazioni a un minimo di 50 ns e un massimo di 90 ns. Quando si usano nodi di temporizzazione dagli ingressi alle uscite di un singolo nodo, essi modellano i ritardi da piedino a piedino del componente che realizza l'operazione.

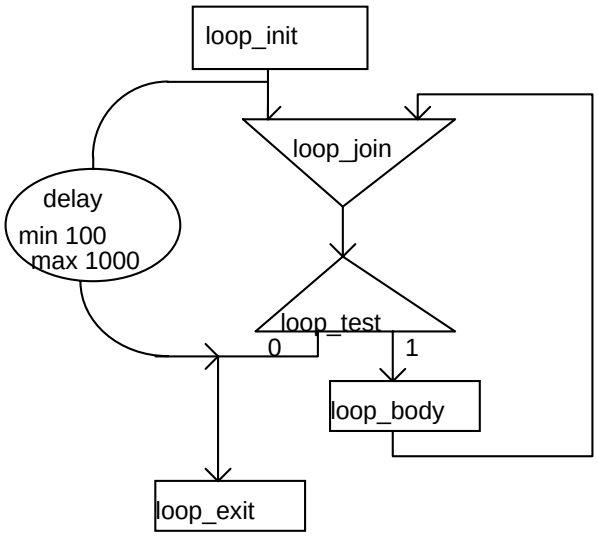


Figura

Si può inserire un nodo di temporizzazione anche fra due archi del ***CFG***. La semantica dei nodi di temporizzazione dipende dallo schema in cui si inseriscono. In un CDFG, un nodo di temporizzazione fra due archi di flusso del controllo dà un vincolo per l'esecuzione di tutti i blocchi di flusso dei dati lungo il percorso di controllo fra origine e destinazione del nodo di

temporizzazione. Fornisce quindi dei vincoli sulle prestazioni di un insieme di diversi blocchi di flusso dei dati o anche dell'intero progetto.

Un esempio che fornisce i vincoli minimo e massimo di ritardo per l'esecuzione di un ciclo, specificato inserendo un nodo di ritardo dall'arco d'ingresso nel ciclo all'arco di uscita dal ciclo, nel grafo di flusso del controllo:



Figura

Nel seguito si fa riferimento allo schema in cui:

- il DFG rappresenta operazioni e dipendenze di dati;
- si affianca al DFG uno schema grafico che indichi diramazioni e cicli.

Estensione:

1) uso di ***data flow graphs*** che rappresentino operazioni, dipendenza e serializzazione dei dati;

Il DFG implica l'esistenza di *variabili*, ognuna delle quali ha un *tempo di vita* - intervallo fra la *nascita* (istante in cui il valore è generato da un'operazione) e *morte* (ultima riferimento del valore come ingresso di un'operazione). Il valore deve essere presente per tutto il tempo di vita ⇒ deve esistere una *memoria* (registro o altro) in cui il valore venga mantenuto.

2) uso di ***sequencing graphs*** che modellano anche il *controllo.* Diramazioni e iterazioni vengono modellate mediante il concetto di *gerarchia.*: Il modello gerarchico consente di rappresentare la *chiamata di modello* (incapsulamento di sottoinsiemi di DFG che possono essere richiamati da più punti).

Un sequencing graph è una *gerarchia di grafi* ("entità") contenente due tipi di nodi - *operazioni* (come in un DFG) e *collegamenti* (link). I nodi-collegamento permettono la connessione di varie entità entro la gerarchia.

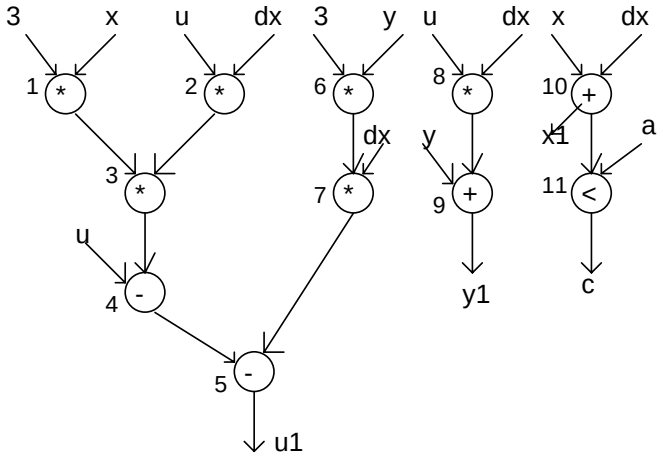
Un semplice DFG (S.G. privo di gerarchia) è *aciclico* - le iterazioni vengono modellate *esternamente* al DFG. Per trasformare un DFG non connesso in un grafo connesso, si introducono nodi *NOP*, (No Operation) - operazioni nulle, eseguite in tempo 0 senza effetti

collaterali. Tutti i *nodi iniziali* dei sottografi disgiunti diventano *successori* di un nodo NOP "sorgente" (nodo 0) tutti i *nodi terminali* diventano *predecessori* di un nodo NOP "pozzo" (nodo *n*).

Si consideri il seguente segmento di programma: (che risolve l'equazione differenziale *y''+3xy'+3=0* nell'intervallo [0,a]; il passo d'integrazione è *dx,* i valori iniziali sono *x(0)=x, y(0)=y, y'(0)=u*):

```
read (x,y,u,dx,a);
repeat (
    x1=x+dx;
    u1=u-(3*x*u*dx)-(3*y*dx);
    y1=y+u*dx;
    c=x1<a;
    x=x1;u=u1;y=y1;
until (c)
```

Si considera il corpo del ciclo; i quattro assegnamenti vengono spezzati in 11 operazioni elementari.



Figura

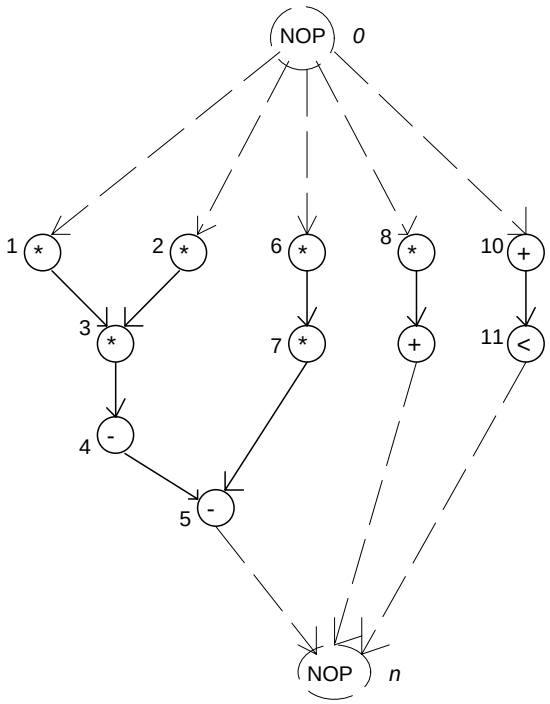


Figura. sequencing graph

Si indica ora la rappresentazione di *model call*, costrutti di *diramazione* e costrutti di *iterazione*.

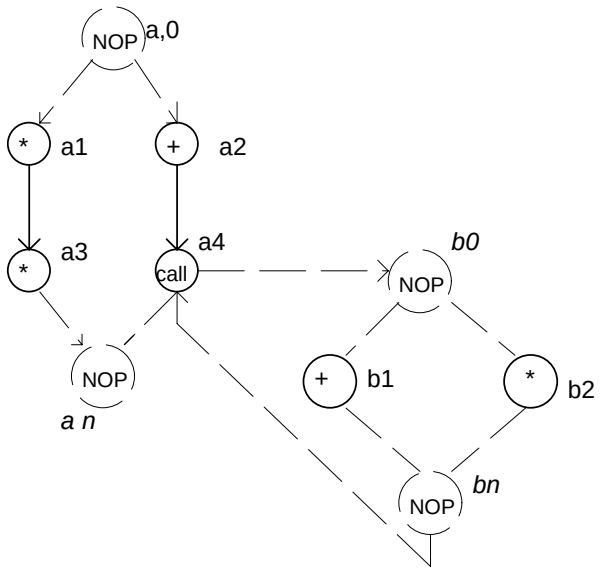
Vertice che rappresenta la model call: è un puntatore a un'altra entità di livello gerarchico inferiore: es.: data la sequenza di codice (scritta in HW C)

```
x=a*b;
y=x*c;
z=a+b;
submodel (a,z);
```

dove

```
submodel (m,n)[p=m+n; q=m*n]
```

Le due entità hanno nodi marcati *ai* (livello superiore) e *bi* (modello): il vertice *a4* è il collegamento che effettua la model call:



Figura

Costrutti di diramazione: modellati da una *branching clause* e da *corpi di diramazione*. Ci sono tanti corpi di diramazione quanti i possibili valori della branching clause, tutti mutuamente esclusivi.

Es.:

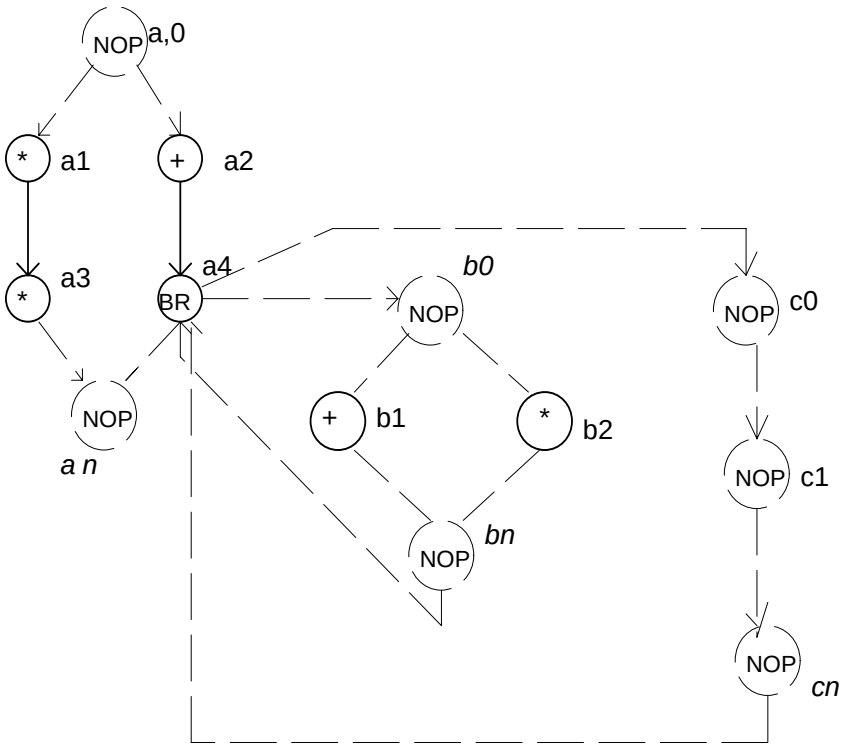
x=a*b;

y=x*c;

z=a+b;

if (z≥0)

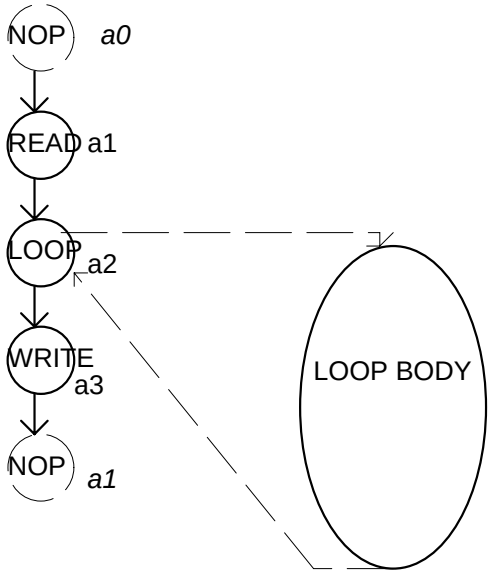
[p=m+n;q=m*n]



Figura

dato che quando la branching clause ha valore falso non succede nulla, il secondo corpo è rappresentato da NOP.

I costrutti iterativi: sono modellati da una *iterative clause* e da un *corpo dell'iterazione*. Considerando l'esempio relativo alla soluzione dell'equazione vista prima possiamo completare la costruzione del sequencing graph integrando il grafo costruito per il corpo con quello che usato per descrivere l'iterazione.



Figura

In generale ogni nodo di un sequencing graph può:

1) essere in attesa di esecuzione;
2) stare eseguendo;
3) aver terminato l'esecuzione.

Un nodo viene *attivato* (fired) quando inizia ad eseguire; può essere attivato non appena tutti i suoi diretti predecessori hanno terminato la loro esecuzione. Attivando il nodo sorgente si lancia l'esecuzione dell'intero grafo.

IL PROBLEMA DEL PARTIZIONAMENTO

Partizionamento è l'operazione che raggruppa oggetti in modo da ottimizzare una data funzione obiettivo rispetto a un insieme di vincoli di progetto. Si applica a vari livelli di astrazione: da quello di layout (per trovare componenti fortemente connessi, che vengono posti vicini per minimizzare area e ritardi di propagazione) a quello di sistema, per dividere un sistema di grandi dimensioni in chip (o, prima ancora, in schede).

Nella sintesi ad alto livello il partizionamento è usato per *scheduling, allocazione, scelta della unità, partizionamento di sistema e di circuito.*

Primo obiettivo: si raggruppano variabili e operazioni in modo che ogni gruppo venga trasposto su un elemento di memoria, un'unità funzionale o un'unità di interconnessione del progetto reale. Il risultato può essere usato per:

3) la scelta delle unità (prima delle operazioni di scheduling e binding) o per l'allocazione. Permette una prima valutazione di massima dell'area richiesta.
4) in relazione allo scheduling: può essere fatto in modo che ogni gruppo di operazioni venga eseguito nello stesso passo di controllo.

Secondo obiettivo: si usa il partizionamento per scomporre una descrizione comportamentale di grandi dimensioni in altre più piccole. Gli scopi sono:

- l'ottenimento di sottoproblemi più trattabili,
- la creazione di descrizioni che possono poi essere sintetizzate con circuiti individuali.

Si considerano innanzitutto alcuni metodi fondamentali di partizionamento, basati su grafi che modellano il comportamento del sistema.

Si consideri la descrizione di un sistema in VHDL comportamentale:

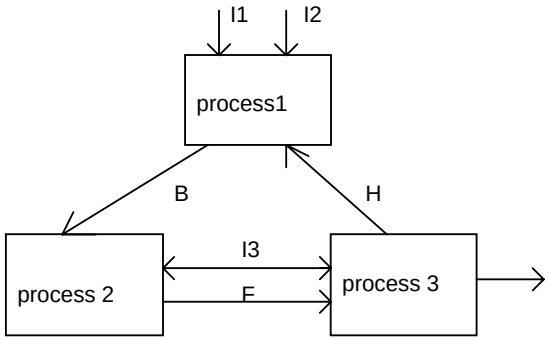
```
entity VHDL EXAMPLE is
    port (I1, I2, I3: in integer;
          O1: out integer;)
    end entity;

architecture BEHAVIOR of EXAMPLE is
    signal B, F, H: integer;
begin
    process
        variable A, C, E: integer;
    begin
        while (I1>0) loop
            if (B>0) then
                B<=C-I2;
            else
                B<=A-I2;
            end if;
            wait until (H>0);
        end loop;
    end process;
    process
        variable D: integer;
    begin
        wait until (B<=0);
        D:=I3+B;
        F<=I3+I1;
    end process;
    process
        variable G: integer;
    begin
        wait until (F>0);
```

```
        O1 <=I3+G;
        H<=I3+I1;
    end process;
end BEHAVIOR;
```

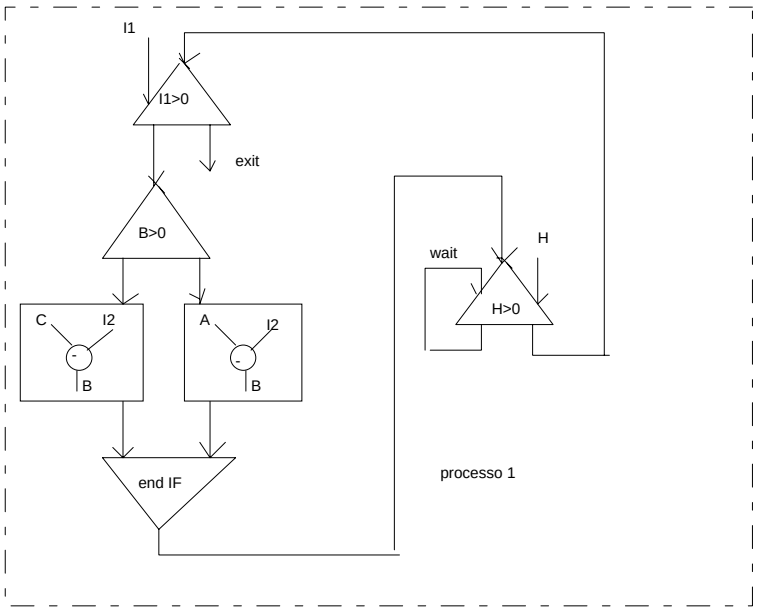
L'architettura descrive tre processi concorrenti: *porte* e *segnali globali* (*I1, I2, I3, O1, B, F, H*) sono usati per la comunicazione fra i processi. Ogni processo può essere visto come un modulo hardware che lo implementa: si può rappresentare un grafo in cui:

- i nodi sono i processi
- i lati rappresentano le variabili globali usate per le comunicazioni fra processi:

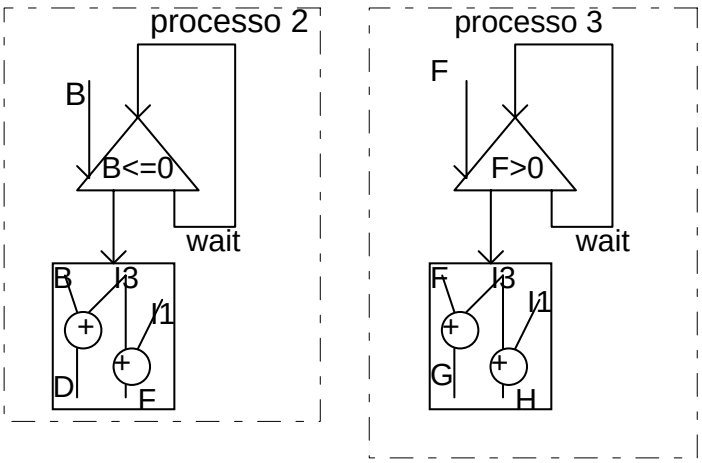


Figura

Infine, si consideri il CDFG:



Figura



Figura

Il partizionamento può essere compiuto con l'obiettivo di ottimizzare le prestazioni (in termini di velocità di esecuzione) o il costo fisico: nel primo caso si raggruppano i nodi sui percorsi critici in modo da minimizzare le comunicazioni (valutate col numero di passaggi di controllo o dati fra i gruppi).

Il partizionamento può generare più percorsi di controllo eseguiti in parallelo, coordinati da un controllore globale o da diversi controllori locali.

Nel secondo caso, i nodi vengono raggruppati in base all'operazione che compiono e si cerca di minimizzare la connettività fra i gruppi in quanto qui rappresenta il numero di connessioni fisiche.

Minimizzare le *comunicazioni* e le *connessioni* sono due finalità diverse: si possono avere comunicazione elevata e connessione bassa (es.: comunicazioni seriali). Il partizionamento mirato alle prestazioni ottimizza l'uso del tempo, quello mirato al costo fisico ottimizza l'uso dei componenti.

Esistono due tecniche fondamentali di partizionamento:

1) metodi costruttivi: il grafo viene partizionato partendo da uno o più nodi "seme" e aggiungendovi altri nodi uno a uno. I metodi sono: selezione casuale, accrescimento dei gruppi e raggruppamento gerarchico.
2) metodi di miglioramento iterativo: si parte da una partizione iniziale e se ne migliorano i risultati spostando man mano oggetti fra i vari gruppi della partizione.

Si considereranno dapprima i metodi costruttivi e poi quelli di miglioramento iterativo.

Selezione casuale

È il metodo costruttivo più semplice, usato spesso anche per generare la partizione iniziale cui applicare poi un metodo di miglioramento iterativo. Si scelgono a caso i nodi, uno per volta, che vengono posti in gruppi di dimensioni fisse finché si raggiungono le dimensioni previste: procedura ripetuta finché si esauriscono tutti i nodi. Si tratta di un metodo molto semplice, che dà risultati spesso modesti.

Accrescimento dei gruppi

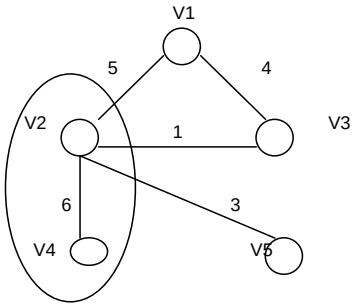
Si parte da un insieme di oggetti non partizionati, che vengono posti in un certo numero di gruppi sulla base di una qualche misura di "prossimità". L'algoritmo di accrescimento è in tre fasi: scelta del seme, selezione dei nodi non ancora inseriti, inserzione dei nodi. I nodi-seme possono essere specificati dal progettista, scelti a caso o scelti in base a una valutazione (es.: dimensioni, numero dei lati di collegamento). Il successivo ordine di collocamento dei nodi non ancora inseriti è determinato da una misura - es. numero di collegamenti fra i nodi già inseriti e quelli non ancora inseriti: il nodo non inserito che ha il massimo valore di prossimità viene inserito nell'opportuno gruppo, e la procedura viene ripetuta fino a quando tutti i nodi non sono inseriti. L'algoritmo facile da realizzare, dà però risultati modesti.

Raggruppamento gerarchico

La misura di prossimità viene valutata a priori per tutti gli oggetti. I due oggetti più prossimi vengono raggruppati per primi e considerati un oggetto singolo per successivi raggruppamenti; la procedura continua raggruppando a ogni iterazioni due oggetti oppure un oggetto e un gruppo già esistente.

L'algoritmo termina quando si è generato un unico gruppo e si è formato un albero gerarchico di gruppi: qualsiasi taglio dell'albero indica insiemi di sottoalberi o di gruppi da estrarre dall'albero.

Es.: sia dato un grafo di cinque nodi con lati pesati: i pesi indicano la "prossimità".



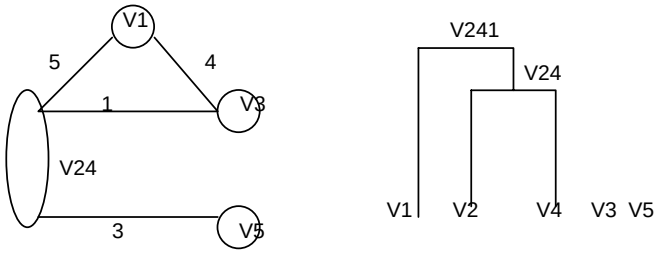
Figura

Si costruisce una matrice di prossimità: V2 e V4 hanno valore massimo di prossimità, vengono raggruppati in V24:

	V1	V2	V3	V4	V5
V1	-	-	-	-	-
V2	5	-	-	-	-
V3	4	1	-	-	-
V4	0	6	0	-	-
V5	0	3	0	0	-

Le misure di prossimità vengono ricalcolate per il nuovo insieme di vertici: la misura è basata sul *massimo* peso fra due nodi o due gruppi.

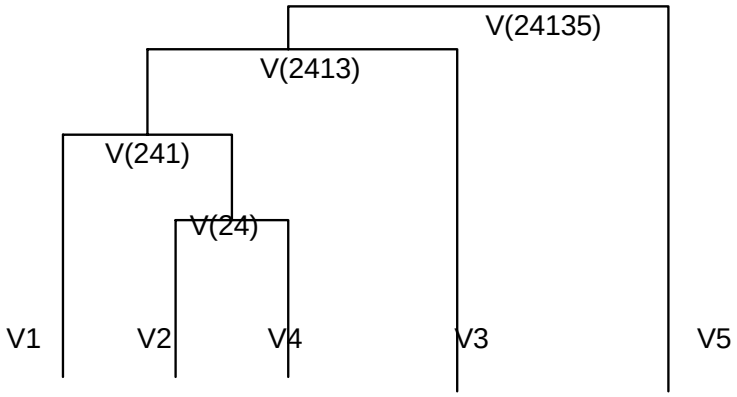
Es.: misura di prossimità fra i nodi V3 e V241 = MAX(4,1)=4. Un passo alla volta, si ottiene un gruppo solo, al quale è associato un albero di raggruppamento.



Figura

	V1	V24	V3	V5
V1	-	-	-	-
V24	5	-	-	-
V3	4	1	-	-
V5	0	3	0	-

Si realizza V241; costruendo un nuovo grafo modificato, si raggruppano poi V2413 e infine si fa il gruppo di tutti i nodi. L'albero dei raggruppamenti finale è:



Figura

Una linea di taglio attraverso l'albero definisce una partizione del grafo. Se la linea di taglio è prossima alle foglie, si definiranno molti gruppi contenenti ognuno pochi nodi vicini; se la linea è prossima alla radice, si generano gruppi grandi contenenti nodi più distanti.

Esistono diverse variazioni di questo tipo di algoritmo; alcune, dette "multi-stadio", consentono di adottare un'intera gerarchia di criteri di prossimità. Gli algoritmi sono più complessi dei precedenti, ma i risultati sono più soddisfacenti.

Le tecniche di **miglioramento iterativo** usano algoritmi molto più complessi dei precedenti. Un algoritmo molto noto è il *min-cut* (o algoritmo di Kernighan-Lin): partendo da una partizione in due sottografi di uguale dimensione, si scambiano due gruppi di nodi in modo da ottenere il massimo miglioramento. La procedura può essere ripetuta (con qualsiasi numero di

partizioni iniziali) finché fra molte partizioni generate si sceglie la migliore. Altre soluzioni scambiano singoli nodi, anziché interi gruppi: si possono raggiungere buoni risultati con complessità computazionale lineare nel numero dei nodi.

Si considerano in maggior dettaglio le fasi di ***scheduling, allocazione*** e ***unit binding***.

IL PROBLEMA DELLO SCHEDULING

Il progettista parte da:

- un sequencing graph
- un insieme di risorse funzionali (eventualmente infinito)
- un insiene di vincoli (es.: latenza)

deve:

1) stabilire in quale istante verrà eseguita ogni operazione;
2) stabilire il mapping operazioni-operatori e variabili-memorie
3) determinare in dettaglio le connessioni nel datapath e la tabella degli stati della FSM di controllo.

Il primo passo è lo ***scheduling***, che riguarda il *dominio del tempo* - si deve stabilire l'istante d'inizio di ogni operazione nel sequencing graph, rispettando i vincoli di precedenza (i nodi NOP chiedono tempo 0 di esecuzione).

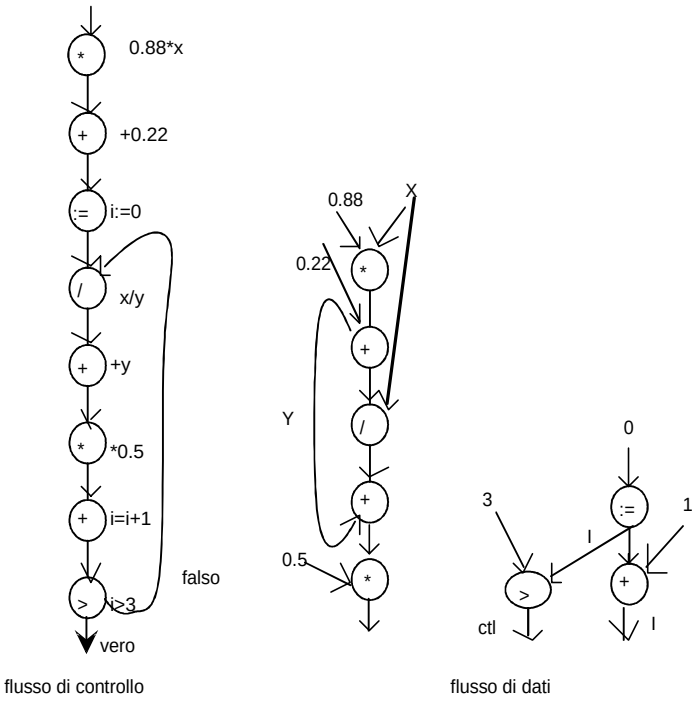
Latenza di uno scheduled sequencing graph è l'intervallo di tempo fra l'istante d'inizio del nodo-pozzo e quello del nodo-sorgente.

Prima di affrontare in termini rigorosi il problema e i vari algoritmi risolutivi, si considera un esempio: si vede come operazioni di ottimizzazione sui grafi e scelta delle unità portano a drastiche modifiche delle prestazioni.

Dato il segmento di programma:

```
Y:= 0.22+0,88*X;
I:=0;
DO UNTIL I>3 LOOP
    Y:=0.5*(Y+X/Y);
    I:=I+1;
ENDDO;
```

una rappresentazione che separa control e data flow graph e rispetta rigorosamente l'ordinamento delle istruzioni nel programma è:

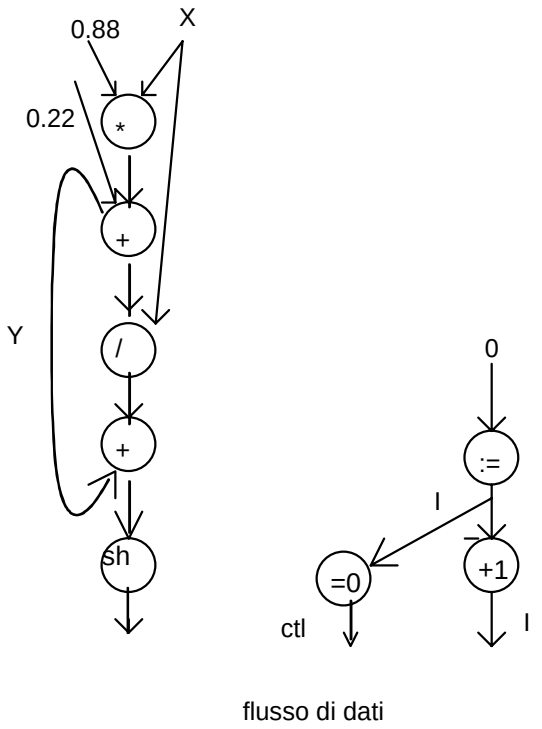


Figura

Prima di affrontare scheduling e allocazione: si analizza l'algoritmo cercando possibili semplificazioni legate alla realizzazione hardware.

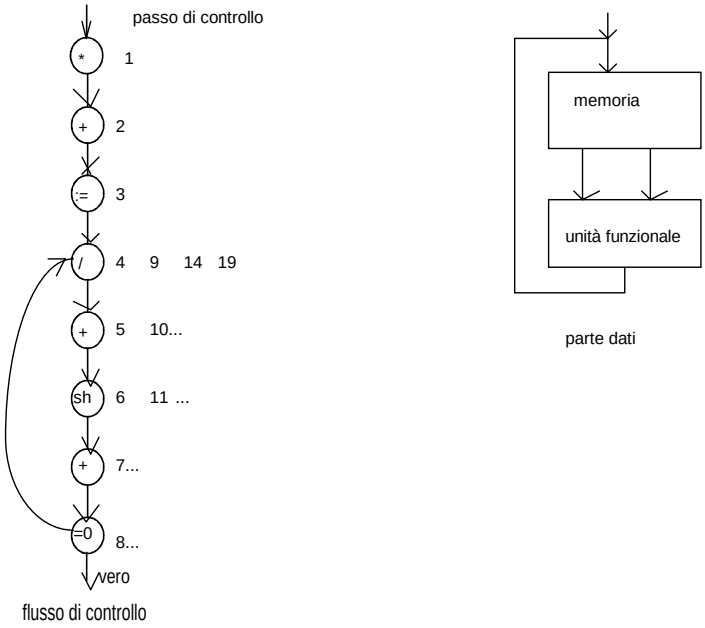
- la moltiplicazione per 0.5 può essere sostituita da uno scorrimento a destra di un bit,
- il criterio di terminazione del ciclo può essere modificato in I=0 se per I si usa una variabile a due bit
- I+1 può essere realizzato con un incrementatore invece che con un sommatore che richieda la costante 1.

Si modifica di conseguenza il DFG (lasciando per il momento immutata la sequenza nel CFG):



Figura

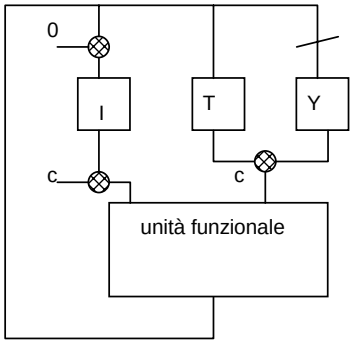
Una realizzazione immediata (scheduling non ottimizzato) usa una sola unità funzionale (ALU) e una memoria: si rappresenta il solo CFG, indicando i passi di controllo per ogni operazione.



Figura

Ogni operazione deve essere prevista (scheduled) in un diverso passo di controllo: la computazione richiede 3+4*5=23 passi di controllo.

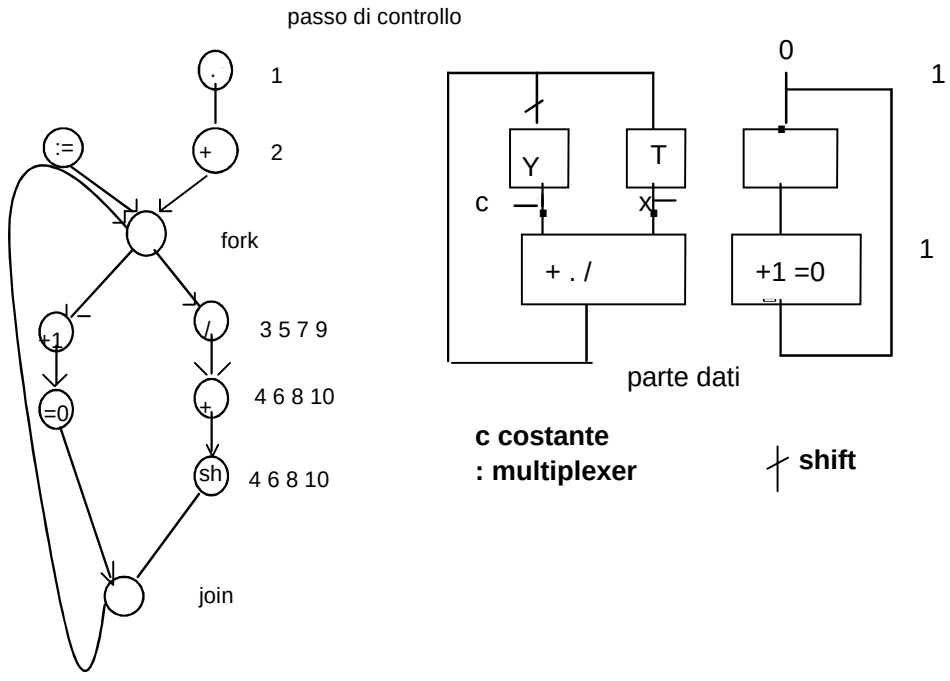
Si sostituisca ora la memoria con due registri per Y e per I e con un registro di servizio T: lo *shift* (6) si fonde con la registrazione del risultato -la computazione richiede 2+4*4=18 passi di controllo.



Figura

passi di controllo:	1	2		
	3	7	11	15
	4	8	12	16
	5	9	13	17

Per accelerare ulteriormente il sistema, si impaccano le operazioni sui passi di controllo sfruttandone il parallelismo e osservando solo le dipendenze fra dati richieste dal DFG. Per delimitare il ciclo, introduciamo due nodi NOP; dal momento che l'operazione di scorrimento non costa passi di controllo, utilizzando due diverse unità funzionali le operazioni richiedono ora 2+4*2=10 passi di controllo.



Figura

Si passa ora ad esaminare in modo rigoroso i vari passi di sintesi.

Il problema dello scheduling

Scopo dello ***scheduling*** è assegnare le operazioni ai passi di controllo in modo da minimizzare una data funzione obiettivo e al tempo stesso soddisfare un insieme di vincoli imposti. La funzione obiettivo può includere il numero di passi di controllo, il ritardo, il consumo, le risorse hardware.

Gli algoritmi di scheduling partizionano il CFG in sottografi, ognuno dei quali deve essere eseguito in un passo di controllo (*passo di controllo* = stato della FSM controllante). Entro un passo di controllo per eseguire ogni operazione si deve utilizzare una corrispondente (separata) unità funzionale. Il numero totale di unità funzionali richieste in un passo di controllo corrisponde direttamente al numero di operazioni previste (scheduled) nel passo.

- realizzazione con pochi passi di controllo (veloce): implica che in ogni passo siano attive più unità distinte (richiede più risorse);
- riduzione del numero di operazioni in ogni passo di controllo: minor costo (meno unità) ma numero di passi di controllo più alto.

Lo scheduling implica quindi un bilancio fra costi e prestazioni.

Si considerino dapprima casi molto semplici in cui:

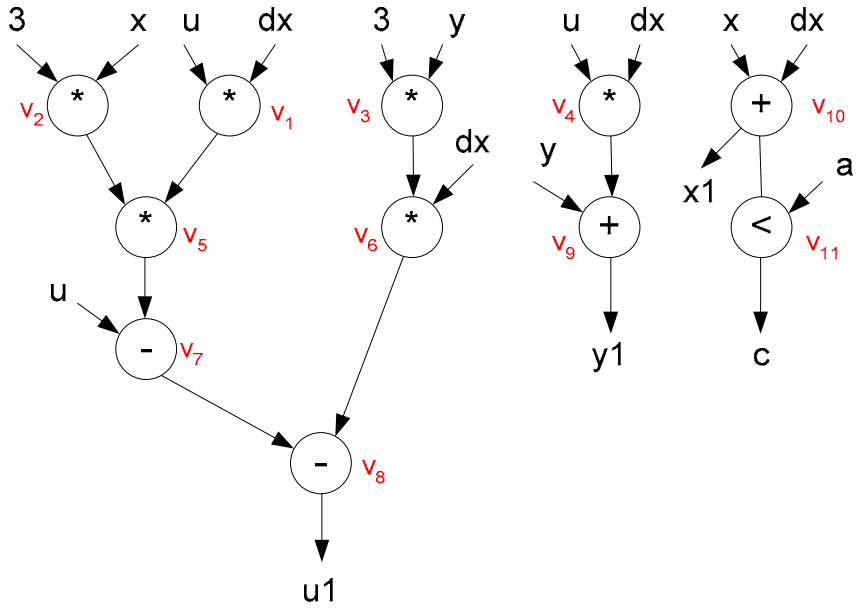
1) le descrizioni comportamentali non contengano costrutti condizionali o cicli;
2) ogni operazione possa essere eseguita esattamente in un passo di controllo;
3) ogni tipo di operazione possa essere eseguito da un solo tipo di unità funzionale.

Data una libreria di unità funzionali con caratteristiche note, definita la lunghezza del passo di controllo, si può:

1) minimizzare il numero di unità funzionali, una volta fissato il numero dei passi di controllo (approccio fixed-latency, o time-constrained);
2) minimizzare il numero di passi di controllo una volta fissato il costo del progetto (es. in termini di unità funzionali e di registri): approccio resource-constrained.

Gli algoritmi operano sui grafi di flusso dei dati (DFG): ogni vertice *vi* rappresenta un'operazione *oi* nella descrizione comportamentale; esiste un lato orientato *eij* da *vi* a *vj* se i dati prodotti da *oi* vengono consumati da *oj* (*vi* è un *predecessore immediato* di *vj*). Ogni operazione *oi* può essere eseguita in *Di* passi di controllo: per il momento, si pone *Di =1*.

Si riprenda l'esempio di DFG relativo alla soluzione dell'equazione differenziale:



Figura

Il DFG mette in rilievo il parallelismo presente nel progetto: esiste una certa flessibilità nell'associare un nodo del DFG a uno specifico passo di controllo. Molti algoritmi di scheduling valutano i limiti estremi entro cui un'operazione può essere assegnata: il primo stato a cui un nodo può essere assegnato viene detto il suo valore ASAP (*As Soon As Possible*): dalla determinazione di tale stato deriva l'***algoritmo di scheduling detto ASAP***.

Si indichi con $Pred_{vi}$ l'insieme dei nodi immediati predecessori di v_i. L'algoritmo assegna un'etichetta ASAP (cioè l'indice del passo di controllo) E_i a ogni nodo del DFG, assegnando così la corrispondente operazione al primo passo di controllo possibile. L'algoritmo ASAP è il seguente:

for each node $v_i \in V$ do
    if $Pred_{vi} = \varnothing$ then
        $E_i=1$;
        $V=V-\{v_i\}$;
    else
        $E_i=0$;
    endif;
endfor;
while $V \neq \varnothing$ do
    for each node $v_i \in V$ do
        if ALL_NODES_SCHED($Pred_{vi}$,E) then
            Ei=MAX($Pred_{vi}$, E)+1;
            $V=V-\{v_i\}$;

endif

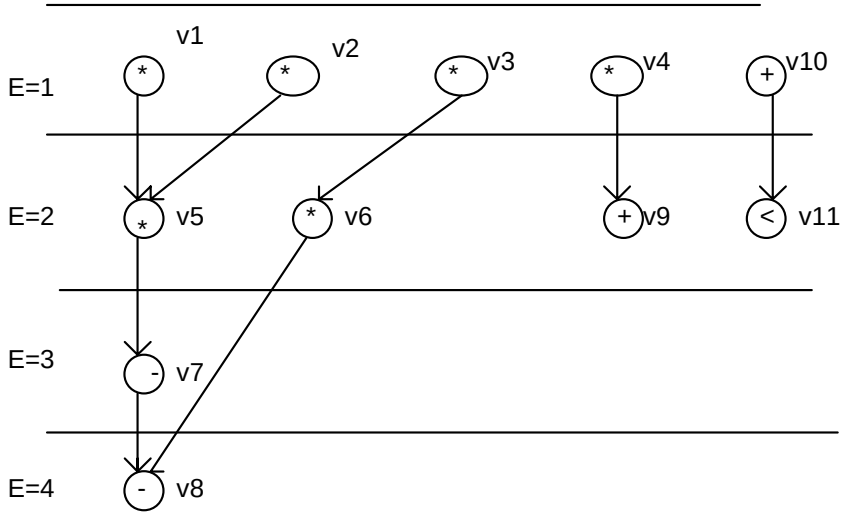
endfor

endwhile

La funzione ALL_NODES_SCHED restituisce valore vero se tutti gli immediati predecessori di v_i sono stati assegnati a un passo di controllo (hanno quindi un'etichetta E non nulla). La funzione MAX fornisce l'indice del nodo col massimo valore di E fra gli immediati predecessori di v_i.

Il ciclo **for** assegna tutti i nodi che non hanno predecessori allo stato *s1* e gli altri nodi allo stato *s0 (indeterminato)*. A ogni iterazione, il ciclo **while** determina quali nodi abbiano tutti i predecessori già assegnati e li assegna al primo stato possibile (si incrementa di 1 il valore restituito dalla funzione MAX nell'ipotesi che ogni operazione abbia un ritardo di un passo di controllo).

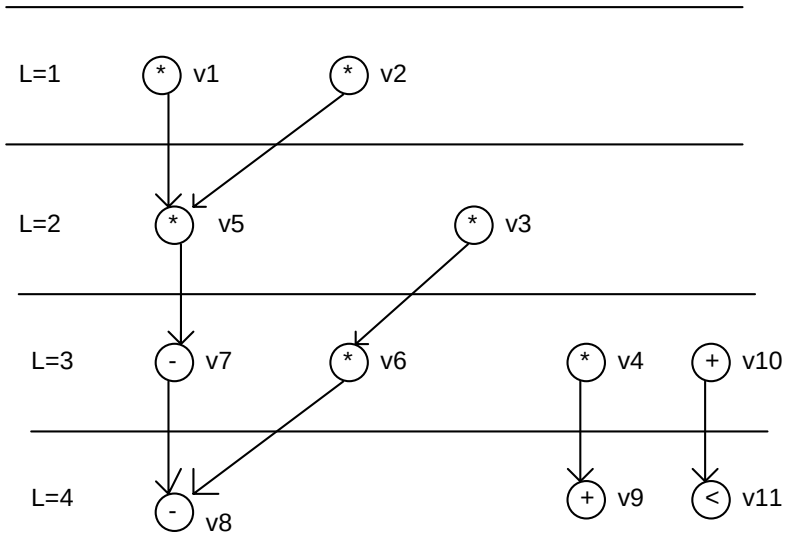
Il risultato dell'algoritmo ASAP sull'esempio dato è il seguente:



Figura

Le operazioni *o1, o2, o3, o4* e *o10* non hanno predecessori, quindi vengono assegnate al passo *s1*, e così via.

Alternativa: ***l'algoritmo ALAP*** calcola per ogni nodo l'*ultimo* stato cui può essere assegnato (soluzione duale della precedente): si assegnano innanzitutto i nodi che non hanno successori, e che vanno sicuramente nell'ultimo stato possibile, poi a risalire con gli stati i cui successori hanno tutti trovato un assegnamento. Per lo stesso esempio, lo scheduling ALAP dà il seguente risultato:



Figura

Ottenuto lo schedule, si calcola il numero di unità funzionali necessarie per realizzare il progetto:

Il numero massimo di operazioni in un qualsiasi stato indica il numero di unità funzionali per quel particolare tipo di operazione. Per l'esempio:

- Soluzione ASAP: il massimo numero di moltiplicazioni in uno stato è 4 (nello stato *s1*) ➔ occorrono quattro moltiplicatori; occorrono inoltre un addizionatore/sottrattore e un confrontatore.
- Soluzione ALAP: il massimo numero di moltiplicazioni previste in un qualsiasi passo di controllo è 2 (stati s_1, s_2 e s_3) ➔ bastano due moltiplicatori: sono inoltre necessari un addizionatore, un sottrattore (questa volta separati) e un confrontatore.

Il numero di passi di controllo richiesto dalle due soluzioni è identico (lo scheduling è fatto a risorse infinite e nessuna operazione potrebbe essere fatta in un passo precedente/successivo a quello in cui è inserita) e determinato dal *cammino "critico"* (più lungo) relativo a dipendenze fra dati nel DFG. Si tratta cioè di soluzioni a *latenza minima.*

Nell'esempio dato la versione ALAP è meno costosa in termini di risorse: fatto del tutto casuale. I due algoritmi presentati *non ottimizzano il numero di risorse* necessarie ma sono molto facili da realizzare.

Le soluzioni Time-constrained

Hanno importanza nel progetto di sistemi dedicati ad applicazioni per sistemi in tempo reale. Es.: sistemi di elaborazione del segnale (DSP) la frequenza di campionamento del segnale d'ingresso vincola il tempo per l'elaborazione su un campione prima che arrivi il campione successivo.

Fissata la frequenza di campionamento, l'obiettivo principale diventa la minimizzazione dell'area. Data la lunghezza del passo di controllo, la frequenza di campionamento può essere espressa in termini di numero di passi di controllo necessari per eseguire l'algoritmo.

Le soluzioni "time constrained" adottano tre tecniche: programmazione matematica, euristica costruttiva, tecnica dei raffinamenti iterativi.

Metodo della Programmazione Intera Lineare (ILP).

Trova uno schedule ottimo mediante una ricerca branch-and-bound che comporta "backtracking": alcune decisione fatte nei passi iniziali possono essere riesaminate man mano che la ricerca progredisce.

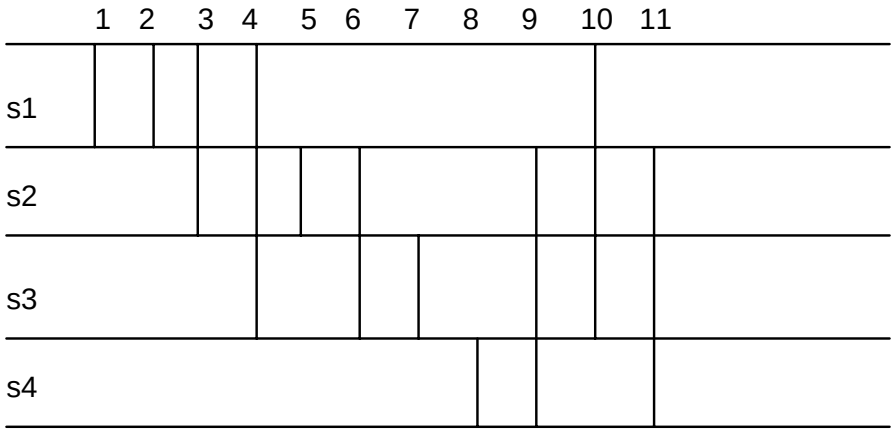
Siano S_{Ek}e S_{Lk}i passi di controllo a cui è assegnata l'operazione o_k rispettivamente con gli algoritmi ASAP e ALAP ($E_k \leq L_k$).

In uno schedule fattibile, o_k deve iniziare l'esecuzione in un passo di controllo che non può precedere s_{Ek} e non può seguire s_{Lk}.

Numero dei passi di controllo fra s_{Ek} e s_{Lk}= *mobility range* dell'operazione o_k:

$$mrange(o_k) = \{s_j | E_k \leq j \leq L_k\}$$

In figura si vede il campo di ogni operazione dell'esempio, calcolati partendo dalle soluzioni ASAP e ALAP:



Figura

Es.: $mrange(o_4) = \{s_1,s_2,s_3\}$ dato che $E_4=1$ e $L_4=3$.

I campi di valori ASAP e ALAP vengono ora usati per lo scheduling usando ILP. Si usa la seguente notazione:

$OP=\{o_i | 1 \leq i \leq n\}$ insieme di operazioni nel *DFG*;

ti= tipo $\{oi\}$;

$T = \{t_k | 1 \leq k \leq m\}$ insieme dei possibili tipi;

OP_{tk} = operazioni in *OP* di tipo t_k;

$INDEX_{tk}$ = insieme degli indici di operazioni in OP_{tk} ;

N_{tk} = numero di unità che compiono operazioni di tipo t_k;

C_{tk} = costo di tali unità;

$S = \{s_j \mid 1 \le j \le r\}$ insieme di passi di controllo disponibili per lo scheduling delle operazioni;

x_{ij} variabili logiche (=1 se l'operaz. *i* è prevista per il passo *j*).

Il problema di scheduling può essere formulato come:

minimizza $\sum_{k=1}^{m} (C_{tk} \times N_{tk})$ *con le ipotesi:*

$$\forall i, 1 \le i \le n, \left(\sum_{Ei \le j \le Li} x_{i,j} = 1 \right);$$

$$\forall j, 1 \le j \le r, \forall k, 1 \le k \le m, \left(\sum_{j \in INDEXtk} x_{i,j} \le N_{tk} \right);$$

$$\forall i, j, o_i \in Pred_{oj} \left(\sum_{Ei \le k \le Li} (k \times x_{j,k}) - \sum_{Ei \le l \le Li} (l \times x_{j,k}) \le -1 \right)$$

La funzione obiettivo minimizza il costo totale delle unità funzionali necessarie.

La prima condizione richiede che ogni operazione sia eseguita in uno e un solo passo di controllo fra i limiti imposti dai due precedenti algoritmi; la seconda condizione garantisce che in nessun passo di controllo compaiano più di *Ntk* operazioni di tipo *tk*. La terza garantisce che tutti i predecessori di una qualsiasi operazione vengano eseguiti in un passo di controllo precedente.

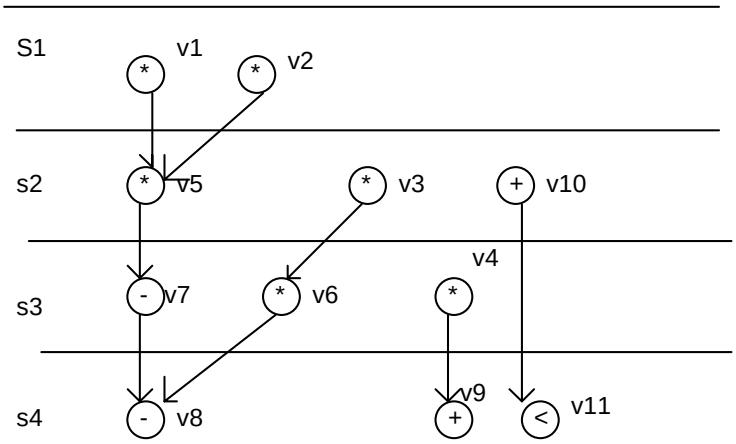
Per l'esempio dato, occorrono quattro tipi di operazioni - quattro tipi di unità funzionali. Siano *Cm, Ca, Cs, Cc,* rispettivamente, il costo di un moltiplicatore, un sommatore, un sottrattore e un confrontatore, la formulazione del problema

assegnare il DFG su quattro passi di controllo diventa la minimizzazione di

$$C_m \times N_m + C_a \times N_a + C_s \times N_s + C_c \times N_c$$

con il vincolo che le operazioni siano eseguite fra i limiti imposti da ASAP e ALAP e dalla struttura del DFG.

Supponendo che addizionatori, sottrattori e confrontatori abbiano costo 1 e i moltiplicatori costo 2, si ottiene la seguente soluzione:



Figura

(in questo caso il costo non è diminuito rispetto alla soluzione ALAP).

Il costo dell'algoritmo cresce molto rapidamente col numero delle variabili, e dipende dalla struttura del DFG. Metodo computazionalmente troppo costoso per casi che non siano molto semplici; si usano varianti in cui si elimina la necessità di backtracking sostituendo tecniche euristiche che assegnano le operazioni una alla volta, invece di tentarne un assegnamento simultaneo.

Metodo euristico "Force-Directed"

Scopo fondamentale: ridurre il numero *totale* di unità funzionali usate. Viene ottenuto distribuendo le operazioni dello stesso tipo uniformemente in tutti gli stati disponibili. La distribuzione garantisce che le unità funzionali allocate in un passo di controllo vengano usate in modo efficiente in tutti gli altri passi di controllo, e alla fine porta a un migliore rapporto di utilizzazione.

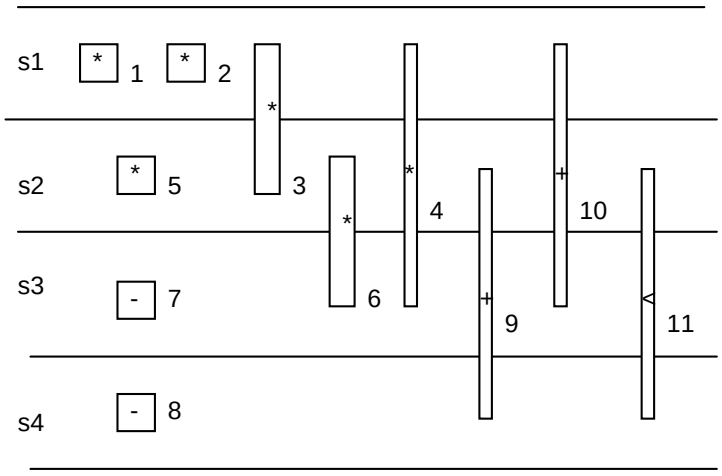
Anche qui si parte dalla valutazione di E_i, L_i per ogni operazione o_i. Si suppone che ogni operazione o_i abbia probabilità uniforme di essere assegnata a un qualunque passo del campo ammissibile, e probabilità zero di essere assegnata a qualsiasi altro passo di controllo.

Per uno passo di controllo s_j tale che $E_j \leq j \leq L_j$ la probabilità che l'operazione o_i sia assegnata a tale stato è:

$$p_j(o_i) = \frac{1}{L_i - E_i + 1}$$

Si fa riferimento al solito esempio. Le operazioni o_1 e o_2 hanno probabilità 1 di essere eseguite in s_1, (hanno ambedue mobilità 0); lo stesso vale per o_5 nel passo s_2, per o_7 in s_3 e per o_8 in s_4.

L'operazione o_3 ha uguale probabilità di essere eseguita in s_1 o in s_2 - quindi probabilità 0.5 in ognuno dei due, etc. Rappresentando le probabilità con rettangoli di ampiezza proporzionale al valore nel passo corrispondente, si ottiene il grafo in figura (a).



(a)

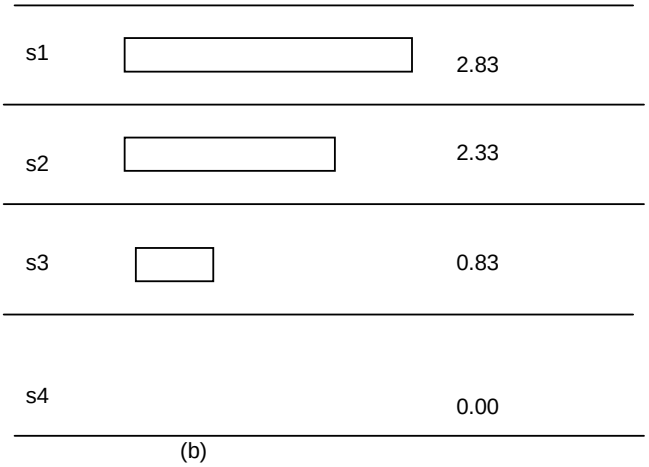
Figura

Si costruisce poi un diagramma a barre *per ogni tipo di operatore*, che rappresenta il costo previsto per l'operatore (EOC) nel rispettivo passo. Per s_j il costo per l'operatore di tipo k è dato da $EOC_{j,k} = c_k * \sum_{i, s_j \in mrange(oi)} p_j(o_i)$

dove o_i è un'operazione di tipo k, c_k è il costo dell'unità funzionale che la esegue. In figura (b) si riporta il diagramma a barre per la moltiplicazione;

$EOC_{1,molt.} = c_{molt.}*(p_1(o_1)+p_1(o_2)+p_1(0_3)+p_1(o_4)) = c_{molt} * (1.0+1.0+0.5+0.33) = c_{molt} * 2.83.$

La barra della moltiplicazione nello stato 1 è quindi proporzionale a 2.83.



(b)

Figura

Le unità funzionali possono essere condivise fra diversi stati: il massimo degli EOC sui vari stati dà una misura del costo totale per l'implementazione dell'operatore dato in tutti gli stati. Diagrammi a barre analoghi si costruiscono per tutti gli operatori.

A questo punto si cerca di bilanciare il valore di EOC per tutti i tipi di operazioni: l'algoritmo che segue è mirato a ottenere questo tipo di distribuzione uniforme: durante l'esecuzione, $S_{current}$

indica lo schedule parziale più recente, S_{work} è una copia dello schedule su cui si tentano assegnamenti temporanei e, in ogni iterazione, *BestOp* e *BestStep* contengono l'operazione migliore da assegnare e il passo migliore cui assegnarla.

Call ASAP(V);

Call ALAP(V);

while there exists o_i such that $E_i \neq L_i$ **do**

MaxGain=-∞

*/*try scheduling all unscheduled ops. to every state in its range*/*

for each o_i, $E_i \neq L_i$ **do**

for each *j*, $E_j \leq j \leq L_j$ **do**

S_{work}=*SCHEDULE_OP*($S_{current}$,o_i,s_j);

ADJUST_DISTRIBUTIONS(S_{work},o_i,s_j);

if COST($S_{current}$)-COST(S_{work})>MaxGain **then**

MaxGain=COST($S_{current}$)-COST(S_{work});

BestOp=o_i; BestStep=s_j;

endif

endfor

endfor

$S_{current}$=*SCHEDULE_OP*($S_{current}$, BestOp, BestStep);

ADJUST_DISTRIBUTIONS($S_{current}$,BestOp, BestStep);

endwhile

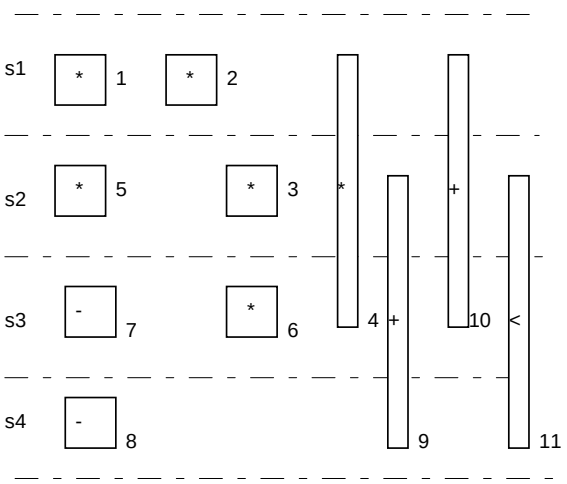
Una volta determinati *BestOp* e B*estStep* per una data iterazione, si modifica lo schedule con la funzione SCHEDULE_OP($S_{current}$, o_i, s_j) che restituisce un nuovo schedule dopo aver assegnato l'operazione o_i allo stato s_j nello schedule corrente.

L'assegnamento di un'operazione a un passo influenza i valori di probabilità delle altre operazioni (per le dipendenze da dati): ADJUST_DISTRIBUTION ricalcola le distribuzioni di probabilità dei nodi predecessori e successori.

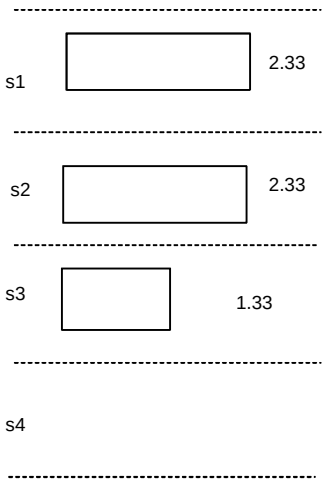
COST(S) valuta il costo per l'implementazione di una schedule parziale S sulla base di una funzione di costo (es.: somma del massimo di tutti gli EOC per ogni tipo di operazione).

Ad ogni iterazione si calcola il costo per assegnare ogni operazione non ancora assegnata entro il suo campo. Si accetta l'assegnamento che dà il costo minimo, si aggiorna $S_{current}$ e, una volta assegnata un'operazione, non la si tocca più nei passi successivi.

Es.: se si assegna l'operazione o_3 al passo s_2 si minimizza COST per la moltiplicazione (Max($EOC_{1,molt.}$) scende da c_{molt} 2.83 a c_{molt} 2.33): si ottengono i seguenti diagrammi modificati:



Figura



Figura

Ad ogni iterazione dell'algoritmo si assegna un'operazione a un passo di controllo; se ci sono due passi di controllo con costi (quasi) identici, l'algoritmo non può effettuare una decisione "ragionata": in alternativa, si possono *escludere* possibili passi di controllo per una data operazione.

L'algoritmo è "costruttivo" perché costruisce una soluzione senza mai fare backtracking. Il risultato può non essere ottimo: si può raffinare il suo valore ricorrendo a tecniche di *raffinamento iterativo* in cui si ammette di riassegnare un'operazione già assegnata, sulla base di opportune strategie (ovviamente, aumenta il tempo di calcolo).

Resource-constrained scheduling

Si deve affrontare quando ci sono vincoli per l'area di silicio (o, il che in genere è lo stesso, per il consumo di potenza - es., applicazioni spaziali, ma anche "smart card...").

Il vincolo può essere dato o in termini di unità funzionali, o in termini di area totale: nel secondo caso, l'algoritmo deve anche determinare il tipo di unità adottato. Scopo: ottimizzare le prestazioni entro il vincolo di area assegnato.

Si costruisce lo schedule un'operazione per volta, in modo da non superare i vincoli di risorse e non violare le dipendenze da dati.

Vincoli sulle risorse: soddisfatti garantendo che il numero totale di operazioni assegnate a un passo di controllo non superi i vincoli imposti (es.: area: si valuta il "floorplan" per l'insieme delle unità: questo è un vincolo che può essere verificato solo dopo che l'intero progetto è stato completato). I vincoli di dipendenza possono essere verificati garantendo che tutti i predecessori di un nodo siano stati assegnati prima di assegnare il nodo.

Algoritmo tipico: List-Based Scheduling Method.

E' uno dei metodi più usati quando ci sono vincoli sulle risorse. E' una generalizzazione di ASAP - in assenza di vincoli sulle risorse, produce gli stessi risultati.

L'algoritmo mantiene una lista di priorità di tutti i nodi "pronti", cioè i cui predecessori sono già stati assegnati. Ad ogni iterazione, si assegnano le operazioni all'inizio della lista fino a quando le risorse disponibili nello stato non sono esaurite: la lista viene sempre ordinata sulla base di una funzione di priorità, che risolve possibili conflitti sull'uso di una risorsa (se c'è un conflitto per l'accesso a una risorsa, si assegna la risorsa all'operazione con priorità massima e si rimandano le operazioni a priorità minore a un passo successivo).

L'assegnamento di un'operazione può rendere "pronte" operazioni che non lo erano, e che vengono inserite nella lista sulla base della funzione di priorità. La qualità del risultato dipende fortemente dalla funzione di priorità.

Si mantiene una lista *PList* per ogni tipo di operazione; le operazioni nelle liste vengono assegnate ai passi di controllo sulla base del numero N_{tk} delle unità funzionali che eseguono l'operazione di tipo t_k.

INSERT_READY_OPS(V,$PList_{t1}$,$PList_{t2}$,...,$PList_{tm}$);

Cstep=0;

while(($PList_{t1} \neq \emptyset$) or...or ($PList_{tm} \neq \emptyset$))**do**

Cstep=Cstep+1;

for funit=1 **to** N_k **do**

if $PList_{tk} \neq \emptyset$ **then**

SCHEDULE_OP($S_{current}$,FIRST($PList_{tk}$),Cstep);

$PList_{tk}$=DELETE($PList_{tk}$,FIRST($PList_{tk}$));

endif

endfor

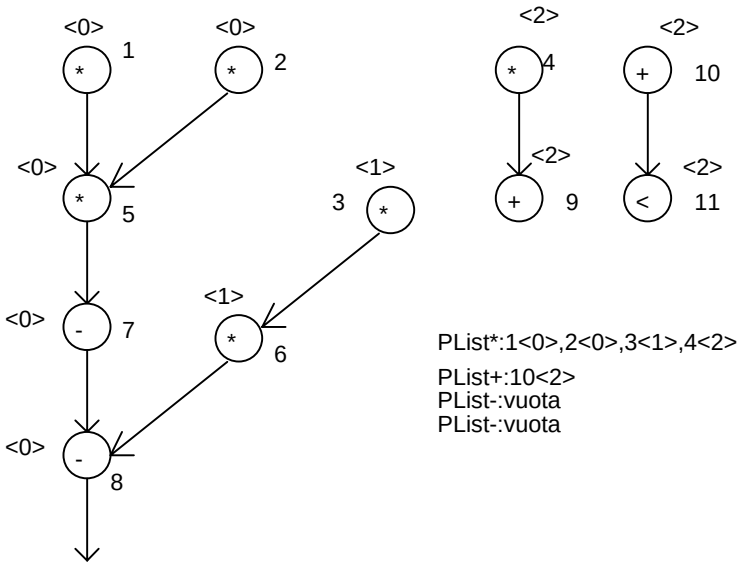
endwhile

La funzione INSERT_READY_OPS analizza l'insieme dei nodi V, determina se ci sono operazioni pronte, le cancella da V e le aggiunge alla lista di priorità relativa al tipo di operazione. La funzione SCHEDULE_OPS($S_{current}$, o_i, s_j) restituisce una nuova schedule dopo

aver assegnato l'operazione o_i allo stato s_j. La funzione DELETE($Plist_k$, o_i) cancella l'operazione indicata dalla lista di priorità.

All'inizio si inseriscono nelle liste di priorità le operazioni che non hanno predecessori; si assegnano le operazioni nelle liste di priorità fino ad aver esaurito le risorse, si rendono pronte nuove operazioni che verranno assegnate alla nuova iterazione, e così via.

In fig. a ogni operazione è contrassegnata col campo di mobilità (indicato fra <>): si assegnano per prime le operazioni con mobilità più bassa, perché ritardarne l'assegnamento aumenta la possibilità di estendere la schedule. (Mobilità = funzione di priorità). Per ogni tipo di operatore si costruisce una lista di priorità in cui si assegna priorità agli operatori pronti con mobilità minore.

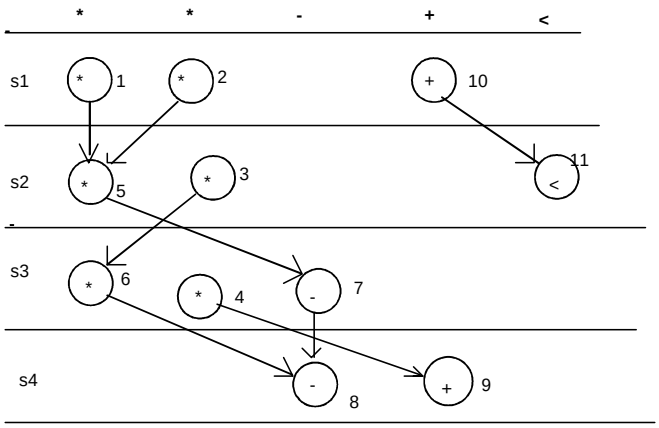


Figura

Prima iterazione: si assegnano le operazioni pronte al passo s_1. Su cinque operazioni pronte (o_1, o_2, o_3, o_4, o_{10}) o_{10} è l'unica addizione: viene assegnata al passo *s1* senza considerare altri fattori.

Nell'ipotesi che siano disponibili due moltiplicatori soltanto, si assegnano allo stesso passo o_1 e o_2, che hanno mobilità più bassa delle altre due. Al passo s_1 sono ora assegnate tre operazioni.

Seconda iterazione: alla lista pronta si aggiungono le operazioni o_5 e o_{11}, i cui nodi d'ingresso sono stati tutti assegnati. La lista pronta consta di o_3 *<0>*, o_5*<0>*, o_4*<1>*, 0_{11}*<2>*: la procedura viene ripetuta e ha termine dopo quattro iterazioni.



Figura

Generalizzazione

Rilasciamo ora alcune delle ipotesi semplificatrici: in particolare si ammettono:

- unità multi-funzione;
- unità funzionali con tempi di esecuzione diversi;
- descrizioni comportamentali che non si limitino a codice lineare.

Unità multifunzionali

Un'unità multifunzionale costa meno di un insieme di untà a funzione singola che coprano lo stesso insieme di funzioni: es.: un addizionatore-sottrattore costa il 20% in più di un semplice addizionatore. $\Rightarrow$ si adatta l'algoritmo di scheduling per tener conto dell'uso di unità multifunzionali.

Se la libreria contiene componenti con uguali funzionalità e caratteristiche diverse (es., area, ritardo) si può usare un algoritmo di scheduling *technology-based*, che opera anche scelte di componenti e realizza il progetto a costo minimo entro i vincoli temporali assegnati.

Es.: si usano componenti veloci per le operazioni sul percorso critico, più lenti per quelle su altri percorsi.

Unità funzionali con ritardi diversi

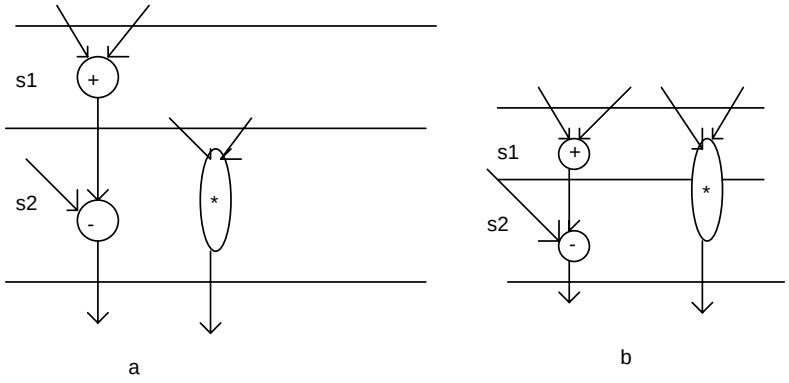
Le unità funzionali reali spesso hanno ritardi diversi, dipendenti dal loro progetto. Non si può quindi supporre che qualsiasi operazione venga completata in un solo passo di controllo: tale ipotesi porterebbe a usare un periodo di clock troppo lungo, per ospitare l'operazione più lenta.

Es.: un moltiplicatore può richiedere il triplo del tempo di operazione di un addizionatore o di un sottrattore - le unità più veloci resterebbero oziose per parte del ciclo di clock.(fig. a).

Si deve ammettere che esistano unità funzionali con ritardo arbitrario, e che queste non vengano più eseguite in un solo ciclo di clock.

Si dimensiona il ciclo clock sulla durata dell'operazione più veloce. Le operazioni più lunghe (operazioni multiciclo) richiedono allora due o più passi di controllo. Si aumenta l'uso delle

unità più veloci, ma occorre introdurre latch d'ingresso sul fronte delle unità multiciclo per conservare gli operandi fino a che i risultati non sono disponibili diversi cicli più avanti (b).



Figura

L'aumento del numero di passi di controllo richiede anche una maggiore complessità dell'unità di controllo. Alternativa per aumentare l'utilizzazione delle unità funzionali: consentire che due o più operazioni vengano eseguite in serie in un solo passo - il **"chaining"**. Si trasferiscono i risultati da un'unità funzionale direttamente agli ingressi della successiva: questo richiede collegamenti diretti fra le unità funzionali (oltre a quelli fra memorie e unità).

Per aumentare il parallelismo di esecuzione, si può anche ricorrere al *pipelining*: quando si usa un'unità funzionale dotata di pipelining, lo scheduler deve calcolare in modo diverso le risorse richieste.

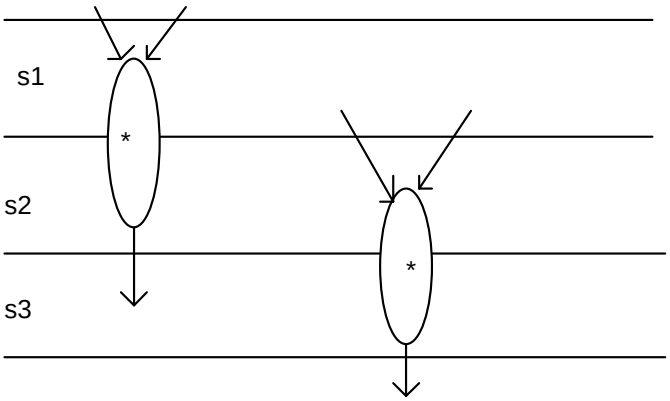
Scheduling dei circuiti pipeline.

La specifica consta di:

- un sequencing graph;
- una frequenza di operazione ("*data rate*").

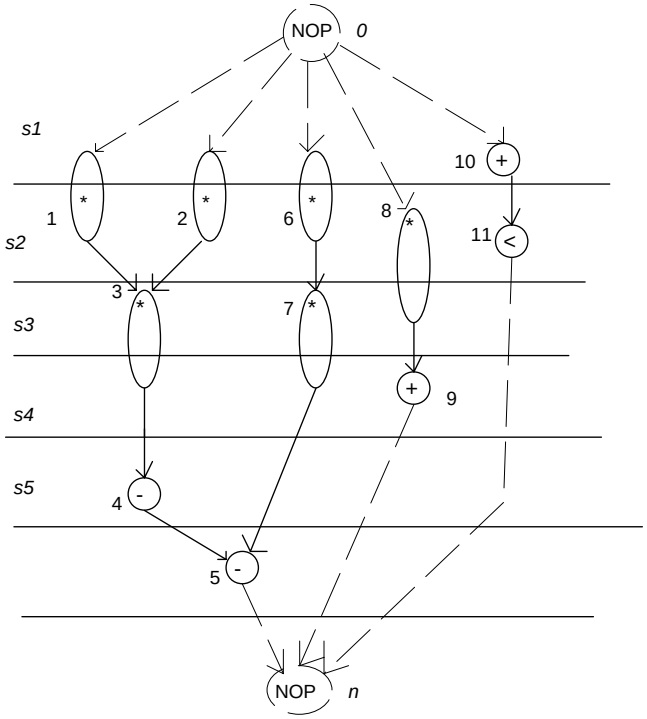
Primo caso: uso di risorse con struttura pipeline, che consumano dati e producono risultati a intervalli di tempo inferiori alla latenza propria dell'intera operazione eseguita. Risorse di questo tipo possono essere *condivise* anche quando l'esecuzione delle due corrispondenti operazioni è sovrapposta.

Es.: uso di un moltiplicatore pipelined a due stadi per l'esecuzione di due moltiplicazioni consecutive: le due moltiplicazioni possono condividere lo stesso moltiplicatore perchè ognuna uso un diverso stadio del moltiplicatore - basta un solo moltiplicatore, invece di due.



Figura

Si riprenda l'esempio dell'integratore, supponendo che i moltiplicatori siano tutti pipeline a due stadi e richiedano tempo doppio rispetto a somme, sottrazione e addizioni:



Figura

Secondo caso: scheduling con vincoli globali di *intervallo di introduzione dei dati* ($\delta 0$) - intervallo fra due esecuzioni successive del nodo sorgente. Si suppone $\delta 0$ costante e inferiore alla latenza complessiva. Per semplicità, si considerano risorse *non pipelined* e grafo *non gerarchico*.

Si noti che (a latenza fissa).

- valore più elevato di $\delta 0$ richiede che più operazioni siano svolte in parallelo - quind maggior numero di risorse (si riduce la "sovrapposizione" possibile fra operazioni successive);

- indicando con *nk* il numero di operazioni di tipo *k*, a, se la frequenza di pipelining è massima ($\delta 0 = 1$) l'uso delle risorse è *ak*=*nk* - tutte le operazioni si sovrappongono e la concorrenza è massima. Quando $\delta 0$ aumenta, assegnando operazioni di un certo tipo a una data risorsa se ne possono serializzare al più

$$\delta 0 \text{ - limite inferiore all'uso delle risorse} = \overline{a_k} = \lceil n_k / \delta_0 \rceil.$$

Si estendono gli algoritmi di scheduling per supportare il pipelining funzionale - in particolare, introducendo il vincolo relativo a $\delta 0$ fra i vincoli su cui si basa il calcolo della funzione priorità.

La trattazione dei costrutti condizionali

Si considerino sequencing graphs contenenti percorsi alternativi - si espandono i nodi di diramazione sostituendole con le corrispondenti entità: la mutua esclusione fra le varie entità dè origine a percorsi alternativi nel grafo. In esecuzione, si esegue un solo percorso, sulla base della valutazione della condizione.

Gli algoritmi list-directed e force-directed possono gestire operazioni mutuamente esclusive se si modifica il calcolo dell'uso delle risorse a ogni passo.

I costrutti ciclici

In molti sistemi (es. filtri digitali) un dato insieme di operazioni viene rieseguito su ogni campione di dati in ingresso ⇒ comportamento descritto mediante un costrutto ciclico. Ottimizzare il *corpo del ciclo* migliora le prestazioni.

Si sfrutta il *parallelismo potenziale fra diverse iterazioni del ciclo*, eseguendole in parallelo. Lo scheduling del corpo del ciclo diventa particolarmente critico.

Es.: ciclo con numero di iterazioni predeterminato e pari a 12: tre soluzioni per lo scheduling:

1. Banale: esecuzione sequenziale. Se ogni iterazione richiede ***b*** passi di controllo, il tempo totale di esecuzione è ***12*b***.

tempo ←b→

1	2	3	4	5	6	7	8	9	10	11	12

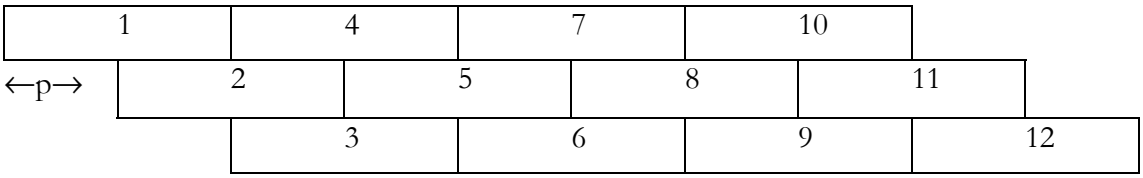
2. Loop Unrolling - si "srotolano" alcune iterazioni del ciclo. Ne risulta un ciclo col corpo più lungoma minor numero di iterazioni - c'è più flessibilità per ottimizzare il corpo (si evita un certo numero di verifiche per la terminazione del ciclo, etc.). Si srotolano tre iterazioni in una super-iterazione che chiede *u* passi di controllo e si eseguono quattro super-iterazioni. Se *u<3b*, il tempo totale di esecuzione è migliorato:

tempo ← u →

1,2,3	4,5,6	7,8,9	10,11,12

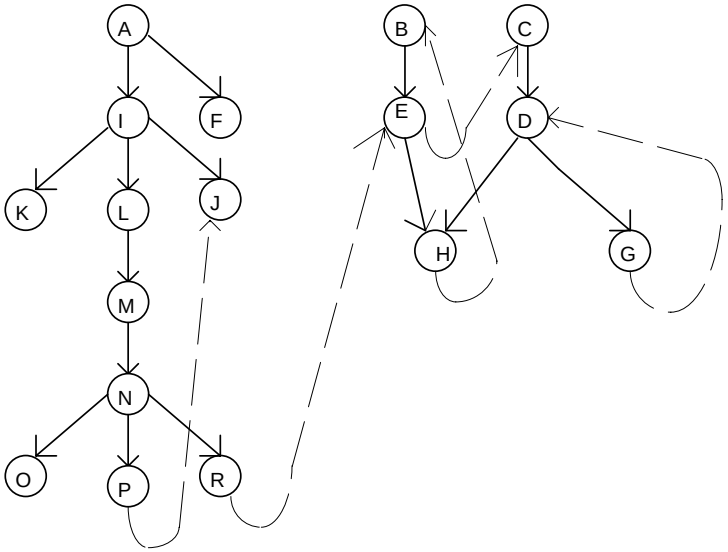
3. Loop folding. Si sfrutta il parallelismo fra iterazioni del ciclo (*intra-loop*) ricorrendo al *loop folding*. iterazioni successive vengono sovrapposte con tecnica pipeline. Se il corpo richiede *m* passi di controllo, si inizia con una nuova iterazione ogni *p* passi di controllo, con *p<m*, sovrapponendo le iterazioni successive. Tempo totale di esecuzione:

*m+(n-1)*p* passi di controllo per *n* iterazioni tecnica applicabile anche quando il numero di passi di controllo non è a priori noto.



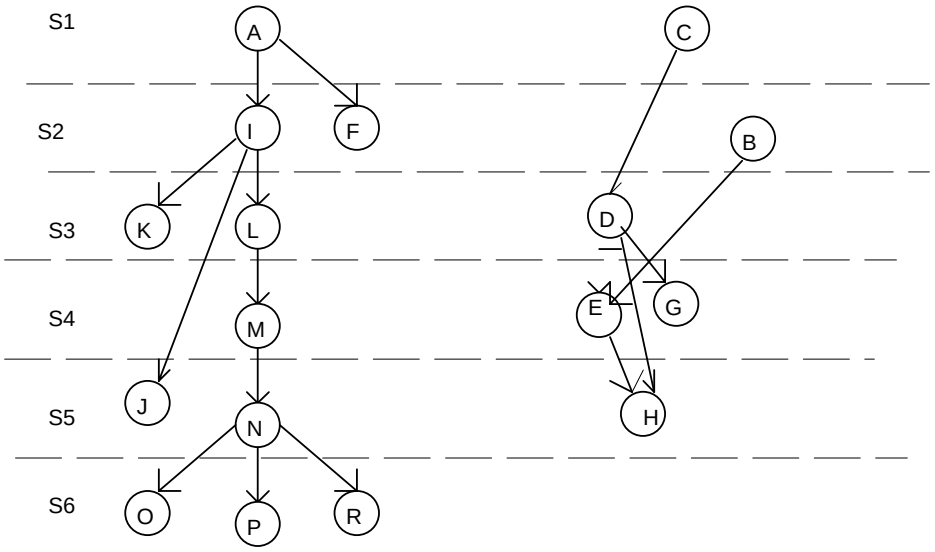
Es.: dato il corpo di un ciclo con 17 operazioni identiche; nel DFG, gli archi tratteggiati indicano che il valore dell'operazione o_i all'iterazione *k-esima* dipendono dal valore dell'operazione o_j all'iterazione *k-1-esima*.

Cifre di merito: utilizzo delle unità funzionali, costo del controllo (numero di parole di controllo "uniche" nell'unità di controllo).



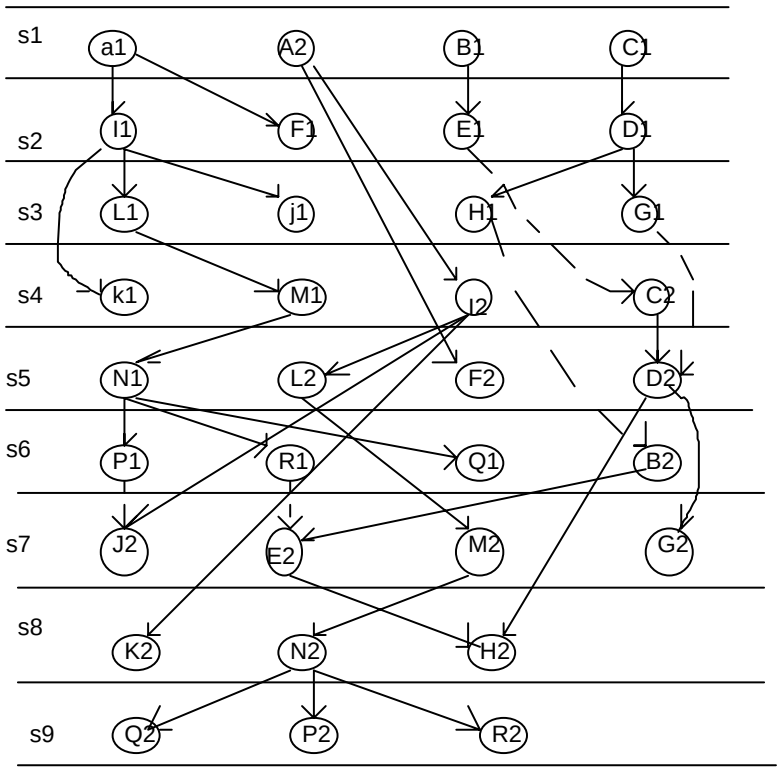
Figura

1. schedule su sei passi di controllo con tre unità funzionali; dato che le operazioni sono 17, lo schedule è ottima in termini di tempo. Costo del controllo: sei parole.



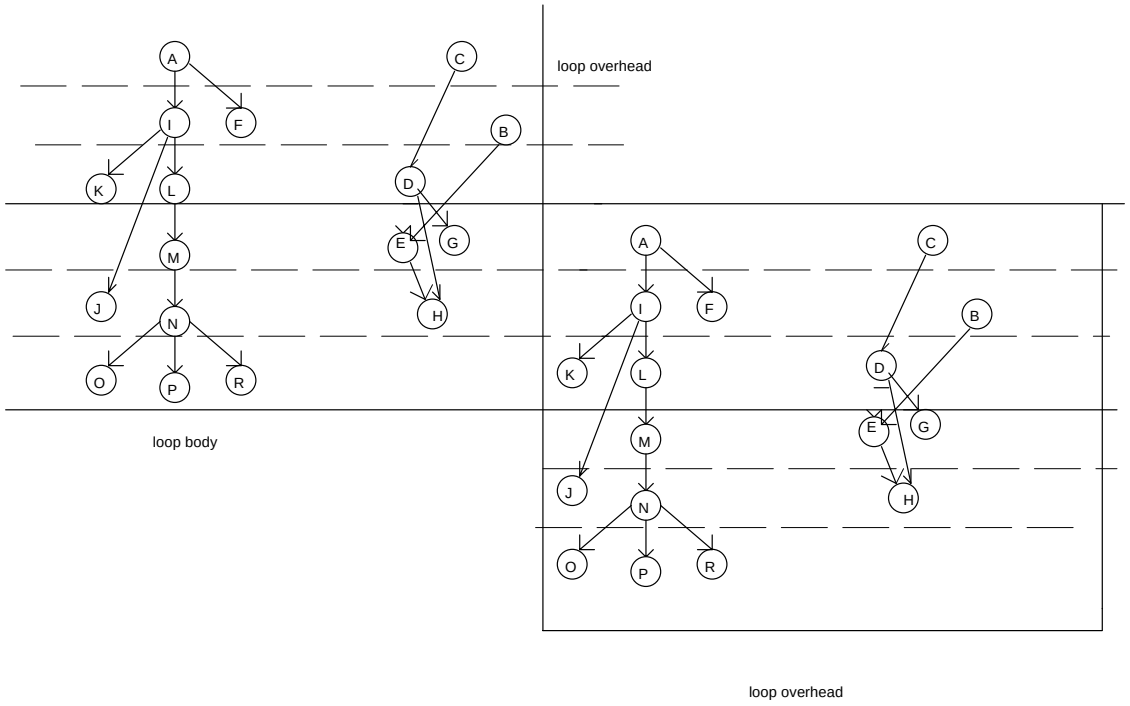
Figura

2.: loop unrolling: si srotolano due copie del ciclo, si usano *quattro* unità funzionali: in nove passi di controllo si eseguono *due* copie del ciclo. Rapporto di utilizzazione hw: (17*2)/(4*9)=17/18 (come nel caso precedente); tempo medio per iterazione: ridotto da 6 a 9/2=4.5 passi di controllo. Costo del controllo: 9 parole.



Figura

3.: loop folding. Si usano *sei* unità funzionali: le iterazioni iniziano a distanza di *tre* passi di controllo (rapporto dettato dalle dipendenze intra-ciclo). rapporto di utilizzazione: 17/(6*3)=17/18; tempo *medio* di esecuzione per un'iterazione pari a 3 passi di controllo, se il numero di iterazioni è abbastanza elevato da annullare lo overhead. Costo del controllo: 9 (tre passi per la testa, tre per il corpo, tre per la coda).



Figura

ALLOCAZIONE E "BINDING" DELLE RISORSE

Lo scheduling assegna le operazioni ai passi di controllo, convertendo così la descrizione comportamentale in un insieme di trasferimenti fra registri e operatori il cui flusso di controllo può essere descritto da una tabella degli stati.

Unità di controllo: sintetizzata a partire dalla sequenza di passi di controllo e dalle condizioni che la determinano.

Datapath: viene derivato dai trasferimenti fra registri assegnati a ogni passo di controllo - fase che costituisce la sintesi del datapath, e coinvolge *allocazione (o condivisione) e binding* delle risorse.

Condivisione delle risorse: assegnamento di una risorsa a più operazioni (scopo principale: riduzione dell'area richiesta).

Binding delle risorse: definizione esplicita di una corrispondenza uno-a-uno fra operazioni e risorse

La fase può essere applicata a DFG già soggetti a scheduling o anche senza che si sia realizzato scheduling (ma con vincoli sulle risorse).

1) per ogni operazione nel CDFG occorre un'unità funzionale capace di eseguirla;
2) per ogni variabile usata su più passi di controllo occorre un'unità di memoria che la registri per tutto il suo tempo di vita;

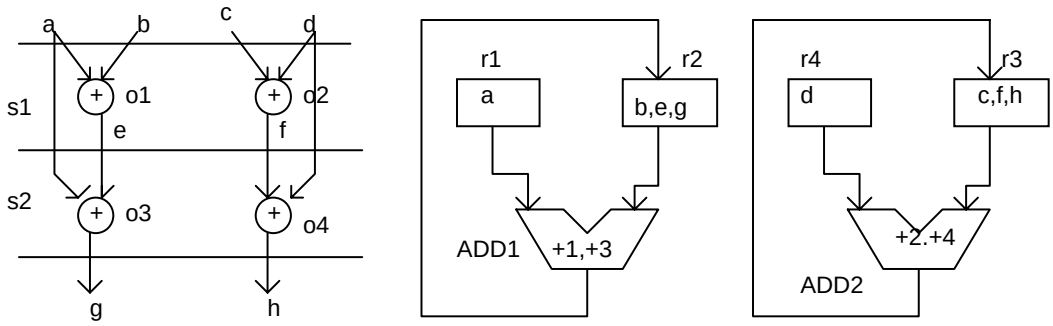
3) per ogni trasferimento di dati nel CDFG, occorre un insieme di unità di interconnessione che lo effettuino.

Nel caso di scheduling con vincoli di risorse: area già determinata dall'uso delle risorse, allocazione e binding servono a definire la rete di interconnessione. Anche le prestazioni effettive finali dipendono dall'uso di registri e connessioni oltre che dallo scheduling.

Un problema ulteriore si ha se si tenta di *coprire* un dato insieme di operazioni con un'unità multifunzionale.

La procedura di unit binding può imporre ulteriori vincoli oltre a quelli originali presentati dal CDFG; ad esempio, il numero di accessi multipli a un'unità di memoria in un passo di controllo è limitato dal numero di porte parallele dell'unità.

Sia dato il seguente DFG (scheduled): si veda il relativo mapping su componenti RT.



Figura

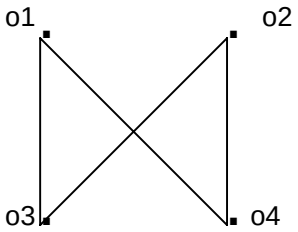
Si siano scelti due addizionatori, ADD1 e ADD", e quattro registri, r1, r2, r3, r4.

Binding delle risorse: due operazioni possono essere associate alla stessa risorsa se non sono concorrenti e se sono dello stesso tipo.

Due operazioni non sono concorrenti se o sono in due diversi passi di controllo, oppure sono in due percorsi di esecuzione alternativi dominati da un costrutto di diramazione.

Si crea un *grafo delle compatibilità* per ogni tipo di operazione che richiede un diverso tipo di operatore; due nodi (=operazioni) sono collegati da un lato se e solo se le due operazioni possono essere realizzate dalla stessa risorsa.

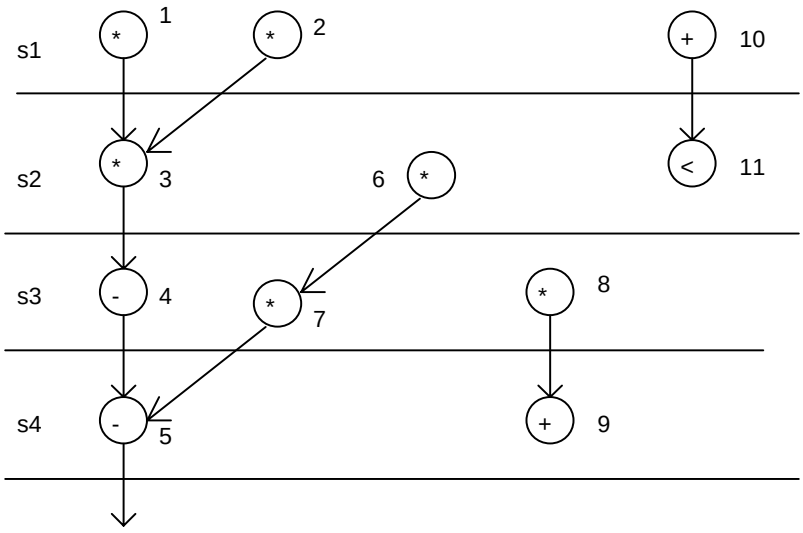
Per l'esempio dei due addizionatori:



Figura

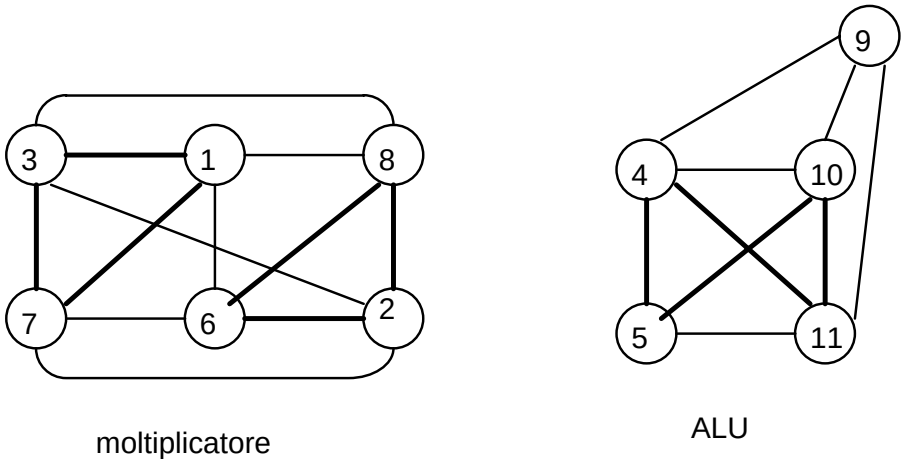
Il binding viene fatto scegliendo un *sottografo completo di massime dimensioni* (una *clique*) e associandolo a una risorsa, fino a esaurire il grafo. Il binding indicato in figura rispetta questa condizione.

Si riprenda il DFG dell'integratore:



Figura

Si supponga di usare come risorse un moltiplicatore e un'ALU capace di fare somme, sottrazioni e confronti. I due grafi delle compatibilità sono:



Figura

Le operazioni 1,3,7 sono allocate al moltiplicatore *m1*, le 2,6,8 al moltiplicatore *m2*; analogamente, le operazioni 4,5,10,11 sono allocate all'ALU 1, la 9 all'ALU 2.

Si considerino ora ***allocazione e binding per i registri***.

Soluzione banale: associare un registro *a ogni variabile*. Chiaramente troppo costosa.

Per ogni variabile si calcola *il tempo di vita*. Per l'esempio iniziale (le due ALU):

a: tempo di vita: s1, s2;

b: tempo di vita: s1;

c: tempo di vita: s1;

d: tempo di vita: s1,s2;

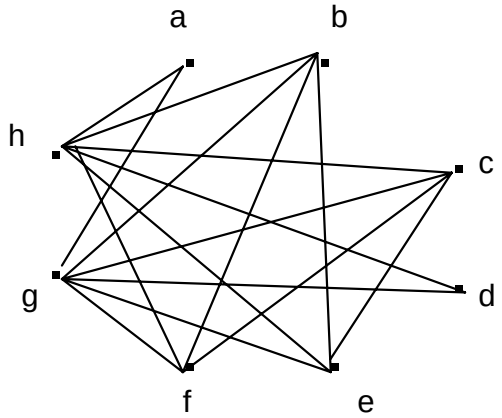
e: tempo di vita: s2;

f: tempo di vita: s2;

g: risultato;

h: risultato;

Le variabili che sono "vive" in intervalli di tempo diversi o in percorsi alternativi dominati da costrutti di diramazione sono *compatibili*. Grafo delle compatibilità:



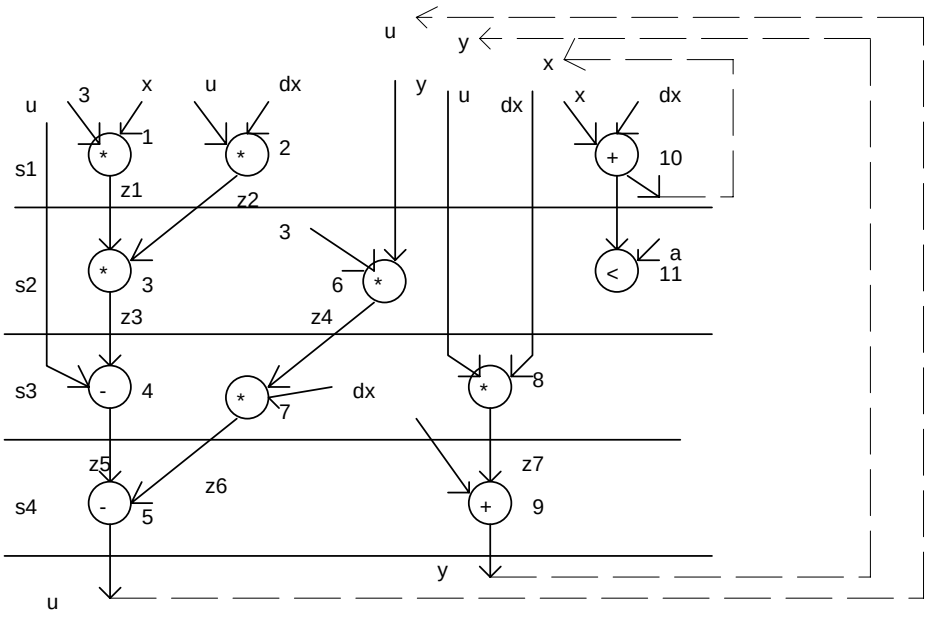
Figura

La soluzione {a}, {beg}, {cfh}, {d} (mostrata in precedenza) è una delle soluzioni possibili corrispondenti al minimo numero di sottografi completi del grafo delle compatibilità.

I registri *r1* e *r2*, in cui *a* ed *e* risiedono, devono essere collegati agli ingressi di ADD1, altrimenti non si potrà eseguire *o3* in ADD1. Allo stesso modo si lavora per ADD2.

Riprendiamo il DFG dell'integratore, mettendo in evidenza

- variabili intermedie (*zi,* i=1,...,7)
- variabili di ciclo (*x, y, u*)
- invarianti di ciclo (*a, 3, dx*).



Figura

Le variabili *x, y, u*: hanno vita su tutti i quattro passi;

- *z1, z2*: vita in s1;
- *z3, z4*: vita in s2;
- *z5, z6, z7*: vita in s3;

Non si considerano le invarianti, che possono essere realizzate come costanti. Le variabili *x,y,u* richiedono altrettanti registri; per le variabili intermedie occorrono tre registri - un binding possibile è, es,

{*z1, z3, z7*}, {*z2, z4, z6*}, {*z5*}.

ARCHITETTURE DI INTERCONNESSIONE NEL DATAPATH

La topologia di interconnessione che supporta i trasferimenti dei dati fra memoria e unità funzionali ha notevole influenza sulle prestazioni del datapath.

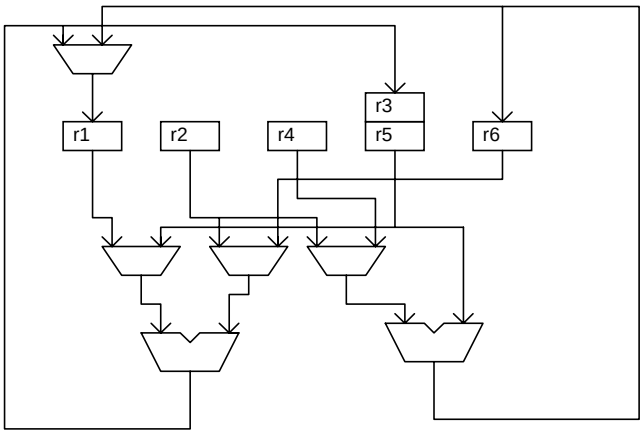
Complessità: definita in base al massimo numero di unità di interconnessione fra qualsiasi coppia di porte di unità di memoria o funzionali.

Ogni unità di interconnessione può essere realizzata con un multiplexer o un bus. Es.: si vedano due datapath (uno basato su multiplexer, l'altro su bus) che realizzano i seguenti cinque trasferimenti:

s1: r3⇐ALU1(r1,r2); r1⇐ALU2(r3,r4);

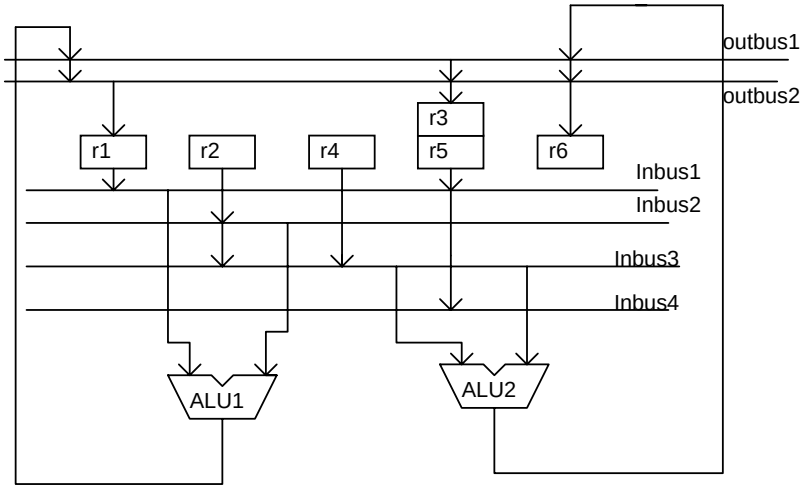
s2: r1⇐ALU1(r5,r6); r6⇐ALU2(r2,r5);

s3: r3⇐ALU1(r1,r6);



Figura

a) soluzione punto a punto



Figura

b) soluzione a bus

Topologia di interconnessione *punto a punto*: fra qualunque coppia di porte delle unità funzionali e/o di memoria c'è una sola unità d'interconnessione. E' la topologia più semplice per la sintesi ad alto livello: si crea un collegamento ogni volta che è necessario. Se all'ingresso di un'unità si assegna più di una connessione, si introduce un multiplexer o un bus.

Per minimizzare il numero di connessioni, si possono raggruppare i registri in banchi con porte multiple, ognuna delle quali supporta accessi di lettura/scrittura. Alcuni banchi di registri consentono letture e scritture simultanee da porte diverse; ogni porta richiede però una circuiteria di decodifica separata.

Si faranno due ipotesi semplificatrici:

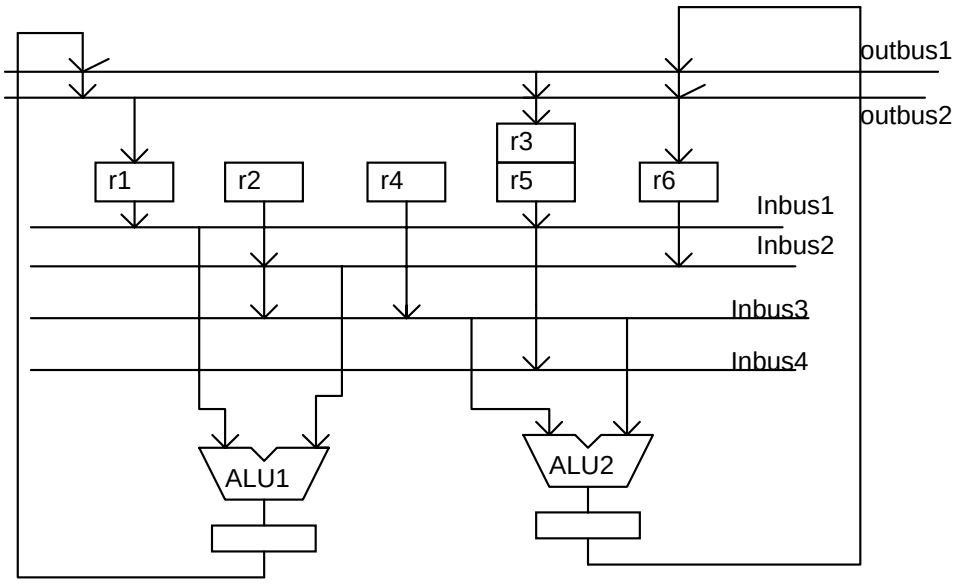
- i trasferimenti fra registri avvengono attraverso unità funzionali;
- non sono permessi collegamenti diretti fra due unità funzionali.

Si considerino i ritardi coinvolti nei trasferimenti da registro a registro dell'esempio in esame: sia t_r il ritardo richiesto per leggere un dato da un registro e propagarlo attraverso la rete d'interconnessione d'ingresso; t_e il ritardo di propagazione attraverso un'unità funzionale; t_w il ritardo per propagare i risultati attraverso la rete di uscita dalle unità funzionali e scriverli nei registri.

I componenti sui percorsi da porte di uscita a porte d'ingresso dei registri sono combinatori: il ciclo di clock ha come limite inferiore $t_r+t_e+t_w$. Temporizzazione relativa delle operazioni nei primi due cicli: riassunta nelle seguente tabella.

Read r1			Read r5			inBus1
Read r2			read r6			inBus2
Read r3			Read r2			InBus3
Read r4			Read r5			inBus4
	Execute			Execute		ALU1
	Execute			Execute		ALU2
		Write r3			Write r1	OutBus1
		Write r1			Write r6	OutBus2
tr	*te*	*tw*				
←	ciclo1	→	←	ciclo2	→	

Si possono inserire latch su ingressi/uscite delle unità funzionali per migliorare le prestazioni del datapath. Inserendoli sulle uscite, gli accessi in lettura e l'esecuzione da parte delle unità funzionali delle operazioni inserite nell'attuale passo di controllo possono essere compiuti contemporaneamente agli accessi in scrittura delle operazioni inserite nel passo precedente. Il tempo di ciclo si riduce a $\max(t_r+t_e,t_w)$.

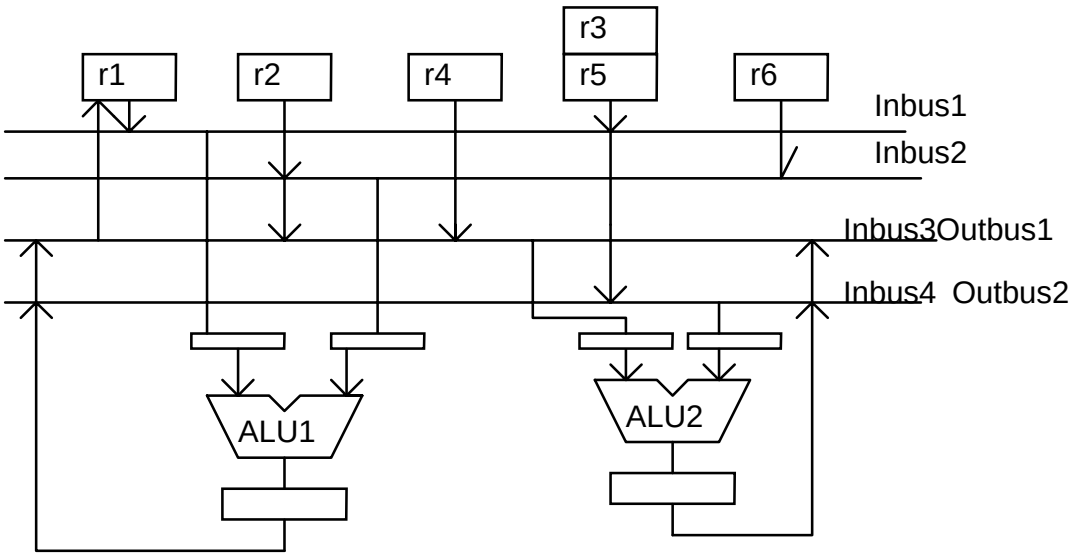


Figura

I trasferimenti fra registri non sono bilanciati: letture ed esecuzione delle operazioni ALU sono eseguite nel primo ciclo, le scritture dei risultati vengono eseguite solo nel secondo ciclo.

Alternativamente, se i latch si mettono solo sugli ingressi delle unità funzionali, gli accessi in lettura delle operazioni inserite nel prossimo passo possono essere eseguiti simultaneamente alle esecuzioni delle operazioni e delle scritture inserite nel passo corrente, e il ciclo diventa max(*tr,te*+*tw*). In ambedue i casi, i banchi di registri e i latch sono controllati da un clock solo.

Interponendo dei latch sia sugli ingressi, sia sulle uscite delle unità funzionali, si possono eseguire simultaneamente operazioni delle unità funzionali e letture/scritture sui registri. I tre componenti combinatori (unità d'interconnessione in ingresso, unità funzionali, unità d'interconnessione in uscita) possono essere simultaneamente attivi. Si ottiene una struttura pipeline: il ciclo di clock è suddiviso in due cicli minori, e l'esecuzione di un'operazione si distribuisce su tre cicli successivi.



Figura

Gli operandi vengono trasferiti dai banchi dei registri ai latch in ingresso alle unità funzionali durante il secondo ciclo minore del primo ciclo; durante il secondo ciclo, l'unità funzionale esegue l'operazione e (entro il termine del ciclo) scrive il risultato nel latch di uscita. Nel primo ciclo minore del terzo ciclo il risultato viene ritrasferito al banco di registri; occorre progettare uno schema di clock a due fasi non sovrapposte. Latch di ingresso e di uscita vengono controllati da una sola fase - termine della lettura e termine di un'esecuzione avvengono simultaneamente. L'altra fase è usata per controllare gli accessi in scrittura al banco di registri.

	Read r1		Read r5		Read r1
	Read r2		Read r6		Read r6
	Read r3		Read r3	Write r3	
	Read r4		Read r5	Write r1	
Exe	cute	Exe	cute	Exe	cute
Exe	cute	Exe	cute	Exe	cute
←Ciclo 1	→	←Ciclo 2	→	←Ciclo 3	→

La sovrapposizione di esecuzione delle operazioni in passi di controllo diversi permette di migliorare l'utilizzazione delle risorse. Il periodo di ciclo si riduce a max(*te, tr+tw)*. Inoltre, reti d'ingresso e di uscita possono condividere alcune unità d'interconnessione (es.: *OutBus1* viene fuso con *InBus3, OutBus2* con *InBus4*). Occorre però definire degli algoritmi di "binding" capaci di scegliere fra un insieme di alternative.

Nel caso si sia adottato il "chaining" di alcuni operatori, occorre introdurre percorsi di bypass intorno ai latch che si trovano su un percorso di chaining.

La sintesi del datapath consta di quattro passi interdipendenti: scelta dei moduli, allocazione delle unità funzionali, allocazione della memoria, allocazione delle interconnessioni.

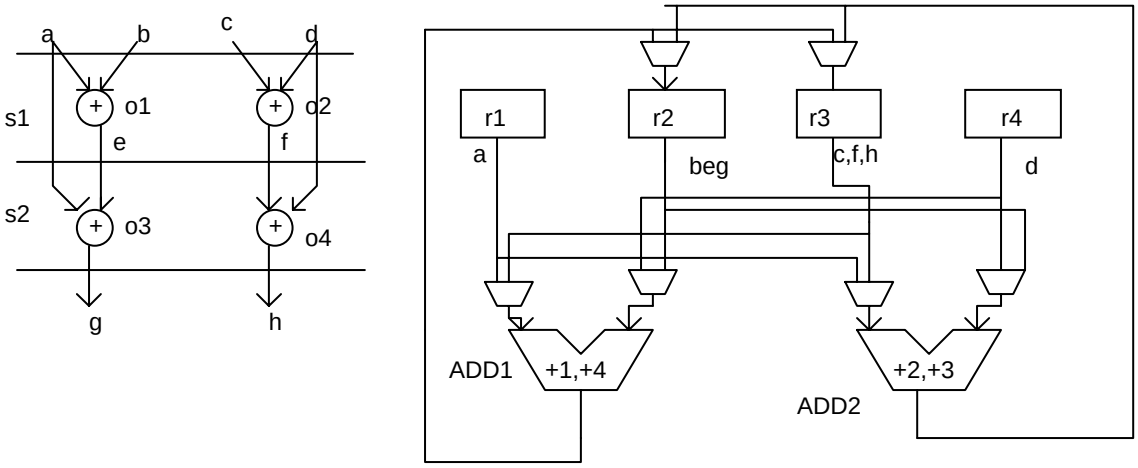
Binding delle interconnessioni.

Due trasferimenti di dati possono condividere un segmento di interconnessione se non avvengono simultaneamente. In questa fase si tende a minimizzare il costo delle interconnessioni massimizzandone la condivisione, pur evitando i conflitti fra i trasferimenti presenti nella descrizione comportamentale.

I tre passi precedenti sono strettamente correlati. Si consideri l'esempio iniziale: la ripartizione delle variabili nel DFG su quattro registri come segue:

$r_1 \leftarrow a$; $r_2 \leftarrow beg$; $r_3 \leftarrow cfh$; $r_4 \leftarrow d$;

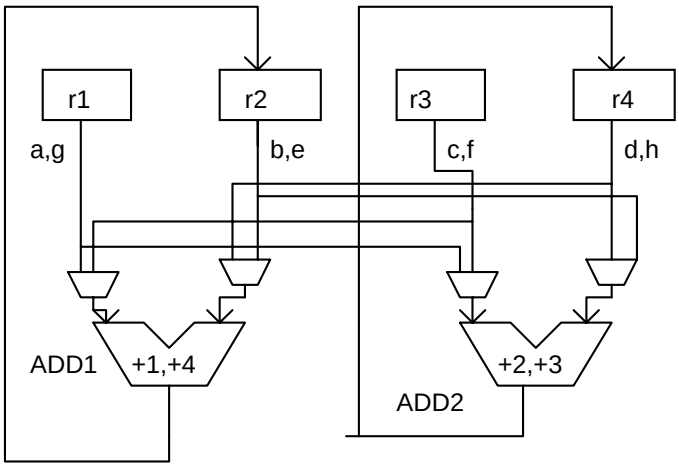
e le due ripartizioni alternative delle operazioni sulle due ALU (rispettivamente, i raggruppamenti o_1,o_2 e o_3,o_4 oppure o_1,o_3 e o_2,o_4) conducono allo schema:



Figura

La soluzione richiede sei multiplexer a due ingressi; si consideri il binding alternativo:

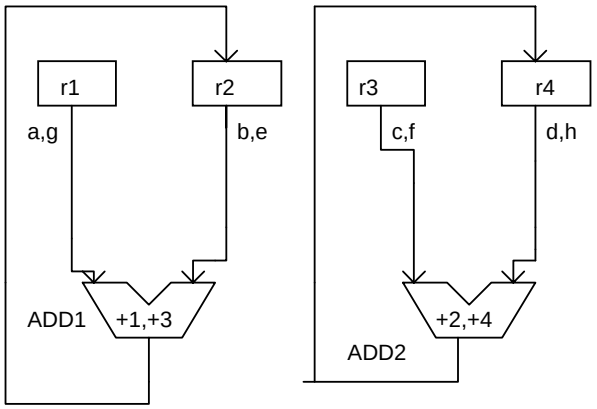
$\{a,g\} \rightarrow$r1; $\{b,e\} \rightarrow$r2, $\{c,f\} \rightarrow$r3; $\{d,h\} \rightarrow$r4: si ottiene:



Figura

Infine, con il binding $\{a,g\} \rightarrow$r1; $\{b,e\} \rightarrow$r2, $\{c,f\} \rightarrow$r3; $\{d,h\} \rightarrow$r4 e

{*+1,+3*}→ALU1, {*+2,+4*}→ALU2 si ottiene:



Figura

Risulta evidente che il binding di memorie e unità funzionali influenza pesantemente quello delle interconnessioni. Nasce il problema di scegliere *quale* binding effettuare per primo: sfortunatamente, non esiste una soluzione completa.

Si può costruire un datapath con un algoritmo "greedy" in cui i componenti vengono assegnati alle operazioni passo-passo. Si parte da un datapath vuoto e lo si costruisce gradualmente aggiungendo le varie unità secondo il bisogno. Per ogni operazione, l'algoritmo cerca di trovare un'unità funzionale sul datapath parzialmente costruito capace di eseguire l'operazione e inutilizzata nel particolare passo di controllo.

- Se ci sono due o più unità in tali condizioni, si sceglie quella che minimizza il costo di interconnessione.
- Se non esiste nessuna unità, se ne aggiunge una scegliendola in libreria. Lo stesso si fa per i registri richiesti dalle variabili.
- L'allocazione di ogni interconnessione si compie non appena si sono bloccate origine e termine di un trasferimento di dati.

Il binding influenza i ritardi sul percorso fra coppie di registri - quindi la lunghezza del ciclo di clock. Si può quindi cercare il binding di unità e registri che minimizzi il ritardo sui percorsi, soddisfacendo tutti gli altri vincoli.

Ulteriori problemi nascono quando invece dei registri si usino *register files* o memorie RAM.

Esistono tecniche per:

- operare simultaneamente scheduling e binding;
- operare allocazione e binding su un DFG non precedentemente soggetto a scheduling;
- applicare allocazione e binding a circuiti di tipo pipeline.

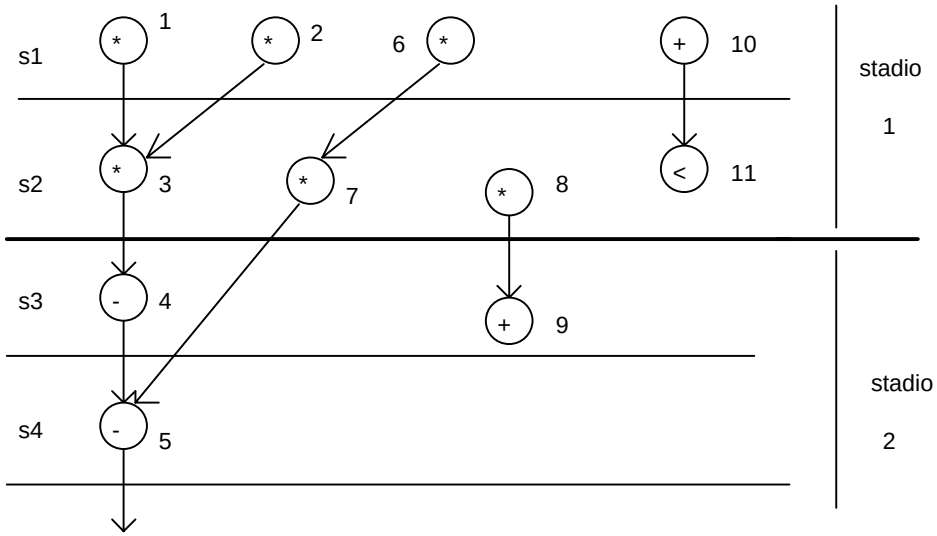
Si considera in particolare l'ultimo caso; l'allocazione è dominata dal *throughput* della pipeline, o (il che è equivalente) dall'intervallo δ_0 di introduzione fra insiemi di dati. Aumentando il throughput si diminuisce la "concorrenza" fra operazioni e si aumentano i conflitti.

Si consideri un grafo su cui si sia operato lo scheduling; si supponga che tutte le operazioni abbiano ritardo di esecuzione unitario. Tutte le operazioni il cui tempo d'inizio sia:

$l + p\delta_0$; $\forall l, p \in Z: 1 \le l + p\delta_0 \le \lambda + 1$ (dove λ è la latenza totale)

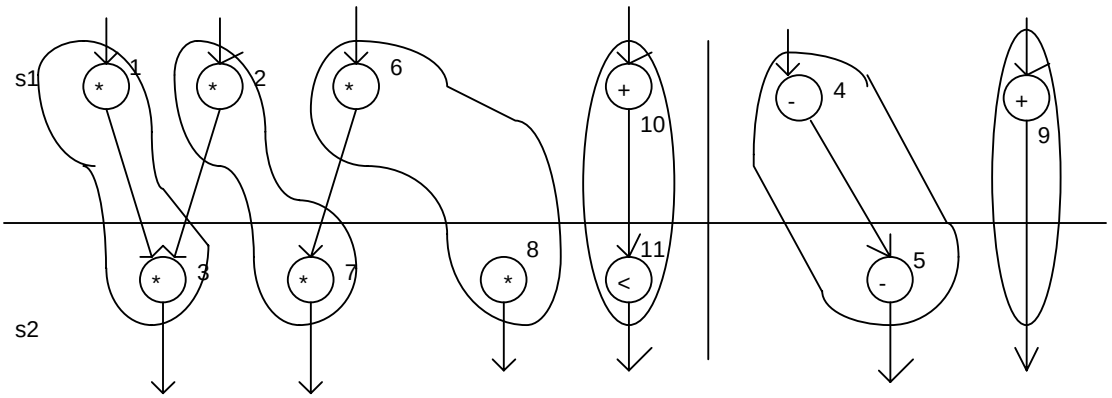
vengono eseguite in modo concorrente. Il grafo delle compatibilità viene costruito tenendo conto di questo fatto.

Si consideri ancora il grafo dell'integratore, supponendo che sia $\delta_0 = 2$ (il ritardo di esecuzione di uno stadio è pari c due passi di controllo).



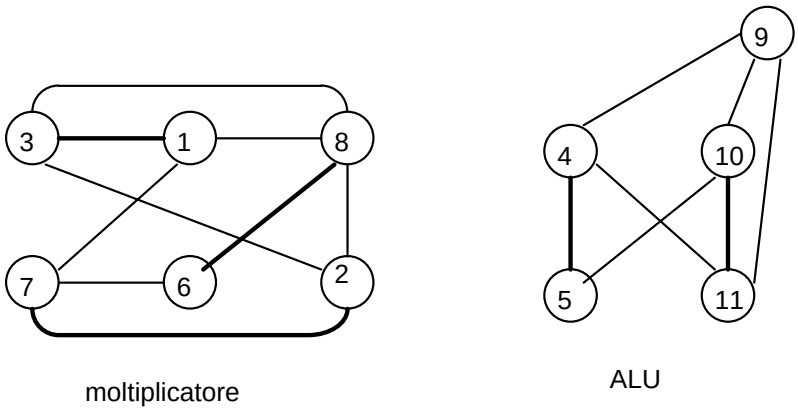
Figura

Si opera un "folding" del grafo come segue, mettendo in rilievo il funzionamento concorrente:



Figura

Lo schema indica che in un passo di controllo s1 si eseguono le operazioni 1,2,6,10 relative all'insieme di dati d'ingresso *i*-esimo e le 4,9 relative all'insieme di dati *i-1*-esimo, e così via. I corrispondenti grafi delle compatibilità risultano modificati:



Figura

Occorrono ora ***tre*** risorse di ogni tipo (di fatto, si è aumentato il parallelismo di operazione).

Problema per le soluzioni pipelined: ***costrutti di diramazione*** - può accadere che si inizi un percorso mentre è ancora in corso di esecuzione il percorso alternativo.

LA SINTESI DELL'UNITÀ DI CONTROLLO

Al termine della sintesi del data path, si conoscono gli stati attraverso cui l'unità di controllo di controllo dovrà evolvere e i segnali che essa dovrà inviare - per ogni stato - per attivare registri, multiplexer etc. (al momento non distinguiamo in categorie tali segnali) (*segnali di attivazione)*. Per sintetizzare l'unità di controllo si possono adottare due diverse soluzioni:

1) unità di controllo microprogrammata. Il microoprogramma è ospitato in una ROM; ogni stato della FSM controllante corrisponde a una parola di microprogramma o microistruzione, costituita dai segnali di attivazione che devono essere inviati al datapath e dall'indicazione sul prossimo stato (indicazione che nel caso generale è l'indirizzo nella ROM della prossima microistruzione; se il CFG è puramente lineare, l'indirizzo può essere generato da un semplice contatore e non occorre indicarlo esplicitamente nel corpo della singola microistruzione);
2) soluzione "hardwired" - la FSM viene sintetizzata come tale, con un circuito combinatorio e opportuni registri.

Ambedue le soluzioni corrispondono al modello astratto della FSM sincrona; la complessità del progetto aumenta se, da sequencing graphs privi di gerarchia nei quali la latenza sia indipendente dal particolare insieme di dati da elaborare, si passa a strutture gerarchiche e a strutture in cui la latenza non sia predeterminata ma dipenda dall'insieme di dati che si sta elaborando

Nei grafi *privi di gerarchia* con ritardi non dipendenti dai dati basta specificare i *segnali di attivazione*. Per i grafi gerarchici, nei quali sono presenti nodi-diramazione e nodi-ciclo, occorre tenere in conto anche i segnali che crrispondono alle *condizioni* che dominano diramazioni o cicli. Infine, se le operazioni hanno una latenza non predeterminata si dovrà mettere in conto anche una *segnale di terminazione*.

Sequencing Graphs non gerarchici con ritardi indipendenti dai dati - Soluzione microprogrammata

Si utilizza una memoria a sola lettura che ha tante parole quanti sono i passi di controllo (λ) e che richiede $\lceil \lg_2 \lambda \rceil$ bit di indirizzamento. L'indirizzamento viene realizzato con un semplice contatore sincrono modulo λ (quindi di $\lceil \lg_2 \lambda \rceil$ bit), con segnale di reset, alimentato dal clock di sistema. Nella soluzione più semplice (la cosiddetta *microprogrammazione orizzontale*) a ogni segnale di attivazione da avviare al datapath si associa un bit della microistruzione; la larghezza n_{act} della parola di ROM può diventare notevole. In alternativa, una solzuione *verticale* provvede a codificare in modo completo tali n_{act} bit; alla riduzione della larghezza della ROM corrisponde l'introduzione di decodificatori (che quindi "allungano il percorso" fra unità di controllo e datapath). In realtà, una soluzione totalmente verticale è raramente possibile, dato che la codifica si può applicare correttamente solo a bit di attivazione che risultino mutuamente esclusivi - in una struttura con un certo grado di concorrenza fra le operazioni, come sono quelle che consideriamo, è difficile che tale mutua esclusione si applichi a tutti I bit di attivazione presenti in un qualunque passo di controllo. La codifica si applica in genere - se mai - a *sottoinsiemi* di bit di attivazione mutuamente esclusivi.

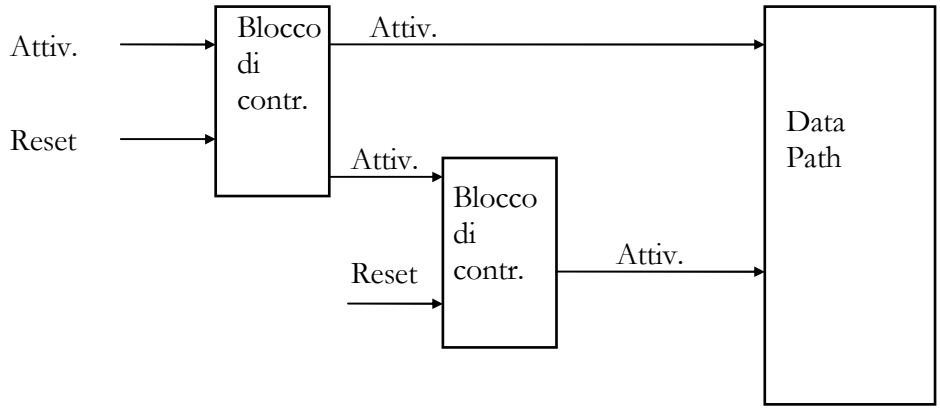
Grafi non gerarchici con ritardi indipendenti dai dati - Soluzione con FSM "hard-wired"

La sintesi è immediata; si tratta di una FSM con λ stati, in corrispondenza biunivoca con i passi di controllo, la cui funzione di uscita fornisce i segnali di attivazione. Dato che la FSM non è dotata di altri ingressi che il segnale di clock, il diagramma degli stati si riduce a un semplice ciclo di lunghezza λ.

Sequencing Graphs gerarchici con ritardi indipendenti dai dati - Soluzione con FSM "hard-wired".

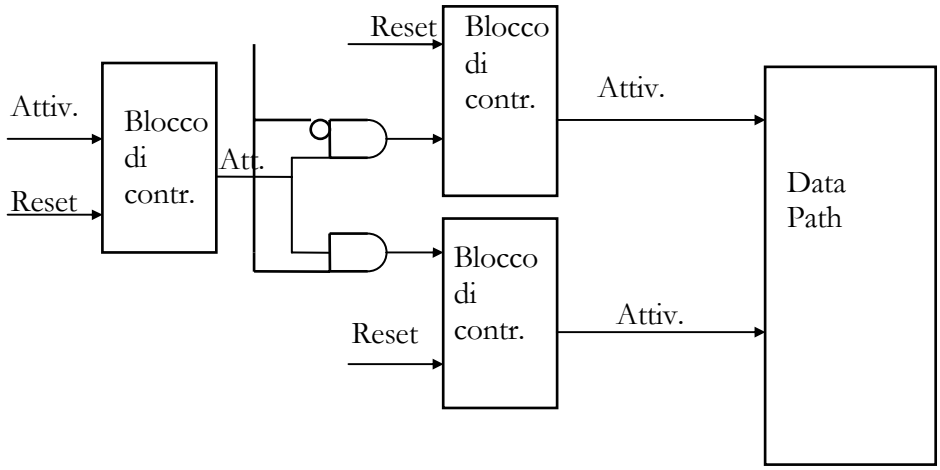
La gerarchia è presente tramite tre possibili casi: l'esistenza di *chiamate a modello*, le *diramazioni*, i *cicli*. Si considerano nell'ordine i vari casi e le rispettive soluzioni.

- Chiamate a modello. Si può pensare che a ogni sottografo corrsipondente a un modello sia associao un'unità di controllo locale (un blocco di controllo) il cui segnale di attivazione viene lanciato dalla chiamata a modello e attiva il contatore dell'unità di controllo del blocco (se si è scelta la soluzione microprogrammata) o le transizioni della maccina a stati nel caso hard-wired. Quando si annulla il segnale di attivazione al blocco, le attività di quest'ultimo vengono fermate e si genera automaticamente un reset che lo riguarda. L'attivazione del blocco di controllo da parte del grafo a gerarchia più elevata (il grafo "chiamante") avviene con un segnale che può essere concorrente con altri segnali di attivazione; di fatto, l'intera eseczuione del blocco di controllo è in genere concorrente con altre operazioni nel grafo chiamante. Si realizza una struttura di macchine a stati gerarchiche; l'esecuzione del nodo-collegamento si traduce inviando al corrispondente blocco di controllo un segnale di attivazione che resta valido fino a quando l'esecuzione del modello chiamato non termina (cioè per tutta la latenza del modello). Durante l'esecuzione del modello chiamato il sistema chiamante può (se del caso) continuare la propria esecuzione. Lo schema della struttura può essere il seguente.



Figura

- Diramazioni. Si possono chiamare come chiamate a modelli alternativi, la cui scelta è determinata dalla valutazione dell'espressione di diramazione. Uno schema di realizzazione può essere il seguente; per garantire che la valutazione dell'espressione di diramazione non cambi nel corso dell'esecuzione del "modello" corrsipondente alla diramazione, è opportuno che il valore venga memorizzato fino a quando tale esecuzione non è terminata.



Figura

- Iterazioni. Il controllo può essere risolto in modo molto simile a quelo adottato per le diramazioni; il corpo del ciclo (che, nel caso di numero di iterazioni indipendente dai dati, verrà eseguito un numero di volte noto a priori) può essere visto come una chiamata a modello ripetuta altrettante volte. Il segnale di attivazione per il corpo del ciclo deve restare attivo per un tempo pari alla latenza del corpo del ciclo moltiplicata per il numero di iterazioni.

Nel caso in cui la latenza del sequencing graph non sia a priori limitata - quando, tipicamente, il numero di iterazioni dei cicli dipende dai dati - occorre che, durante l'ultimo passo di controllo dell'esecuzione dell'ultima iterazione, venga generato un *segnale di completamento* usato per la sintesi della *macchina a stati finiti gerarchica* che costituirà il controllore del sistema. La sintesi

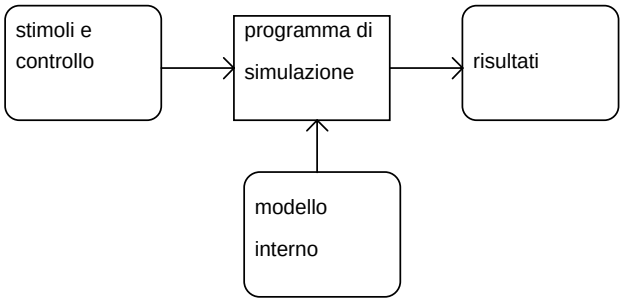
diventa in questa caso notevolmente più complessa che nei casi precedenti; una soluzione può essere quella di realizzare dei "grappoli" (*clusters*) di sottosistemi a latenza predeterminata.

C a p i t o l o 5

LA SIMULAZIONE LOGICA

INTRODUZIONE

La simulazione può costituire una forma di *verifica del progetto* basata sull'uso di un modello del sistema progettato; il programma di simulazione elabora una rappresentazione degli stimoli d'ingresso e determina l'evoluzione nel tempo dei segnali del modello. (La *verifica* di un progetto mira a controllare che il progetto realizzi il comportamento dato nelle specifiche, includendo sia la funzionalità che la temporizzazione).



Figura

Se si usa la simulazione, la verifica viene compiuta confrontando i risultati ottenuti per simulazione con quelli previsti nelle specifiche. Si fa qui riferimento alla *simulazione logica*, che può essere usata anche per verificare che il comportamento del circuito sia:

- corretto indipendentemente dallo stato iniziale;
- insensibile a date variazioni dei ritardi dei componenti;
- privo di corse critiche, oscillazioni, condizioni d'ingresso "illegali" etc.

Altre applicazioni della simulazione nel progetto mirano a:

- consentire la valutazione delle alternative di progetto (cioè l'esplorazione dello "spazio di progetto);
- consentire la valutazione di modifiche proposte per un progetto già completato;
- completare la documentazione del progetto stesso.

La simulazione spesso *sostituisce il prototipo hardware* del sistema (che tradizionalmente veniva realizzato nella fase di progetto) con *un modello software*, più facilmente realizzato ed analizzato. (A volte, per sistemi di particolare complessità, è stata usata anche come passo intermedio per *mettere a punto un prototipo effettivo* del sistema dato).

La verifica di un progetto basata su simulazione consente inoltre di:

- controllare condizioni di errore;
- variare i ritardi in modo da controllare condizioni "di caso pessimo";

- verificare i valori attesi secondo le specifiche dell'utente;
- inizializzare il circuito simulato in uno stato arbitrario;
- controllare in modo preciso la temporizzazione di eventi asincroni (es.: interrupt);
- fornire un ambiente per la successiva *verifica dei guasti* (testing) del circuito simulato.

Vale la pena di notare che non tutte queste operazioni potrebbero, in genere, essere compiute sul sistema fisico (o comunque non potrebbero essere compiute in modo economicamente accettabile); quindi la simulazione è indispensabile per identificare il comportamento del sistema anche in tali situazioni.

Quando si voglia operare una verifica del progetto basata su simulazione, si presentano tre problemi correlati:

- come generare gli stimoli d'ingresso;
- come verificare che i risultati ottenuti siano corretti;
- come garantire (o in quale misura) che gli stimoli applicati forniscano una verifica "completa".

Gli stimoli d'ingresso sono abitualmente organizzati in una sequenza di *casi di prova* (*test*), ognuno dei quali deve verificare un certo aspetto del comportamento del modello. I risultati sono considerati corretti quando corrispondono a quelli previsti nelle specifiche. È importante rilevare la differenza fra *generazione dei test per la verifica del progetto* (lo scopo è l'identificazione degli errori di progetto) e *generazione dei test per l'identificazione dei guasti fisici in un sistema prodotto* (lo scopo è verificare l'integrità fisica del prodotto). Molto spesso, in ambedue i casi si opera a livello logico; infatti, la maggior parte dei guasti fisici possono essere modellati mediante guasti logici, il cui effetto sul comportamento del sistema è ben definito. Si può basare il test per un certo guasto sulla differenza di comportamento fra il sistema in cui il guasto è presente e il sistema sano.

Mentre in genere i modelli di guasto logico rendono i guasti *numerabili*, e la *qualità del test* (cioè dell'insieme di stimoli adottati per il collaudo) può essere valutata come rapporto fra numero di guasti rilevati dal test e numero totale di guasti nel modello (*copertura di guasto*), al contrario lo spazio degli *errori di progetto* non è ben definito, e l'insieme degli errori di progetto non è numerabile: è quindi impossibile sviluppare algoritmi di generazione dei test per la verifica del progetto o definire misure rigorose della qualità per i test stessi. La verifica del progetto mediante simulazione ha gravi limiti. la produzione degli stimoli d'ingresso è una procedura euristica, basata sull'intuito e l'esperienza del progettista; se un sistema "passa" la verifica, si è dimostrato solo che il progetto è corretto rispetto ai casi di test applicati - cioè si dimostra una *correttezza parziale* (e non è in genere possibile dimostrare la completezza del test). Nonostante questi limiti, la simulazione è uno strumento molto utile ed efficace per il collaudo di un progetto (di fatto, l'unico di uso comune oggi).

TIPI DI SIMULAZIONE

I simulatori vengono classificati sulla base del modello interno che elaborano. Una prima classificazione distingue:

1) Simulatori *compiler-driven*: *eseguono* un modello in codice compilato, che può essere stato generato da un linguaggio HDL, oppure da un normale linguaggio di programmazione

o anche da un modello strutturale. Gli stimoli d'ingresso costituiscono i dati per l'esecuzione del codice.

2) Simulatori *table or event driven*: *interpretano* un modello sulla base di una struttura dati, prodotta da un modello HDL o da un modello strutturale. L'interpretazione del modello è guidata ("controllata") dagli stimoli applicati, e provoca una serie di chiamate a routines che implementano operatori o componenti primitivi.

Durante la simulazione di un circuito, i segnali il cui valore cambia in un dato istante si dicono *attivi;* il rapporto fra il numero di segnali attivi e il numero totale di segnali nel circuito si indica come *attività* del circuito stesso. In genere, in un circuito di medie-grandi dimensioni, l'attività è fra 1 e 5 per cento: questo fatto è la base di tecniche dette *activity-based simulation*, che simulano solo la parte attiva del circuito.

Con **evento** si indica il cambiamento di valore di una linea di segnale. Se si ha un evento sulla linea *i*, gli elementi componenti del circuito che ricevono *i* in ingresso si dicono *attivati.*

Con **valutazione** di un elemento componente del circuito (o, il che è lo stesso, delle sue linee di uscita) si indica la determinazione del relativo valore di uscita**;** la simulazione *activity-based* valuta solo gli elementi attivati. La valutazione di alcuni degli elementi attivati può condurre alla modifica del valore delle rispettive uscite, generando così nuovi eventi. La simulazione activity-based si dice anche **simulazione event-driven**.

Per propagare gli eventi lungo le interconnessioni fra elementi, il simulatore richiede un modello strutturale del circuito che consenta di identificare gli elementi attivi e di proseguire la propagazione solo attraverso di essi: per questo motivo, i simulatori event-driven sono in genere di tipo *table-driven*.

La simulazione compilata mira prevalentemente alla *verifica funzionale* e non tratta la temporizzazione del circuito. È quindi applicabile a reti sincrone, la cui temporizzazione viene verificata separatamente. Il passaggio del tempo è invece il punto focale della simulazione event-driven, che può trattare modelli temporali accurati e può quindi gestire sia sistemi asincroni sia *ingressi in tempo reale* - ingressi i cui tempi di variazione sono indipendenti dall'attività del circuito simulato ma legati esclusivamente ad eventi esterni.

Spesso si combinano i due tipi di simulazione - un algoritmo event-driven propaga gli eventi fra i componenti, e i componenti attivati vengono valutati da modelli in codice compilato.

Il ***livello della simulazione*** corrisponde al livello dei modelli usati. Si può avere una simulazione:

- a livello di trasferimento fra registri (RTL): il sistema è descritto mediante la struttura di interconnessione fra componenti modellati come registri e unità funzionali;
- a livello funzionale: il sistema è modellato come interconnessione di blocchi funzionali primitivi. Si può ricadere nel caso precedente, se la complessità dei blocchi è di livello RT;
- a livello di porta logica;
- a livello di transistore (si considera ancora una schematizzazione di tipo logico: nel caso di componenti MOS, il transistore viene modellato come un interruttore - in questo caso, si parla spesso di *livello di interruttore, o switch*);

- a livello misto - ad esempio, alcuni componenti (che siano stati già simulati separatamente in modo soddisfacente) vengono rappresentati a livello RT, altre parti del sistema vengono modellate a livello di porta logica..

Se si opera esclusivamente su reti combinatorie, la simulazione logica può essere effettuata utilizzando I due valori logici 0 e 1 e la condizione d'indifferenza *x*. Quando si passa alla simulazione di reti sequenziali, nasce però un ulteriore problema: lo stato iniziale di una macchina sequenziale, al momento dell'accensione, non è sempre noto (a meno che non sia stato realizzato un apposito circuito di reset). Si deve allora applicare una sequenza di inizializzazione che porti la macchina in uno stato noto di "partenza".

Per trattare lo stato iniziale incognito, la simulazione logica adotta un valore logico indicato con *u* (unknown) e detto *valore logico incognito*. Questo valore è elaborato durante la simulazione unitamente ai valori logici binari, grazie a un'opportuna estensione della logica booleana basata sui seguenti punti:

1) *u* rappresenta un valore nell'insieme (0,1); analogamente, si trattano i valori 0 e 1 come gli insiemi (0) e (1), rispettivamente.
2) l'applicazione di un operatore booleano *B* fra *p* e *q*, dove $p,q \in \{0,1,u\}$, è considerata come un'operazione fra gli insiemi di valori che rappresentano *p* e *q* ed è definita come l'insieme unione dei risultati di tutte le possibili operazioni *B* fra i componenti dei due insiemi.

Così:

$$AND(0,u) = AND(\{0\},\{0,1\}) =$$
$$\{AND(0,0),\ AND(0,1)\ \} = \{0,0\} = 0$$

$$OR(0,u) = OR(\{0\},\{0,1\}) =$$
$$\{OR(0,0), OR(0,1)\} = \{0,1\} = u$$
$$NOT(u) = NOT(\{0,1\}) = \{NOT(0),\ NOT(1)\} =$$
$$\{1,0\} = \{0,1\} = u$$

Le tabelle delle verità di AND,OR,NOT per la logica a tre valori vengono ottenute estendendo quelle abituali, come segue:

AND	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

OR	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

NOT	0	1	u
	1	0	u

La procedura per determinare il valore di una funzione $f(x_1, x_2, \ldots, x_n)$ per una combinazione di valori d'ingresso 0,1, e *u* si articola sui seguenti passi, estendendo quella identificata per valutare una funzione partendo dalla sua rappresentazione cubica:

1) si forma il cubo ($v_1,v_2,\ldots,v_n|x$), dove *x* è la condizione d'indifferenza;
2) servendosi dell'operatore intersezione modificato, si interseca questo cubo con i cubi primitivi di f. Se si trova un'intersezione consistente, nella posizione più a destra si trova il valore di *f*; altrimenti, *f*=*u*.

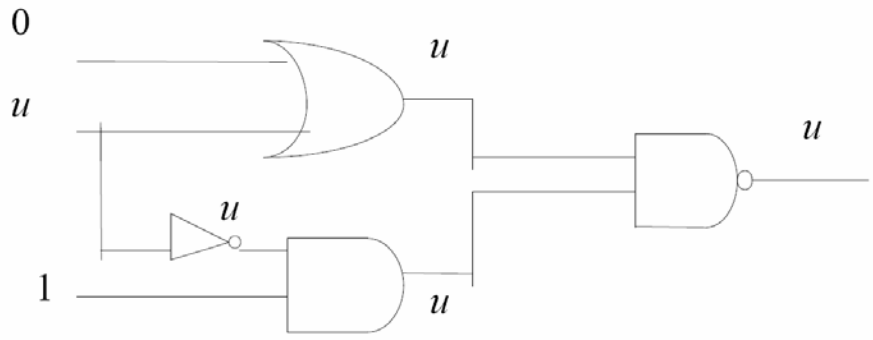
È necessario modificare l'operatore intersezione, in modo da gestire anche il valore incognito:

∩	0	1	*x*	*u*
0	0	∅	0	∅
1	∅	1	1	∅
x	0	1	*x*	*u*
u	∅	∅	*u*	*u*

Si noti che, poiché una *x* in un cubo primitivo indica una condizione d'indifferenza, un valore incognito è *consistente* con un valore *x* in un cubo primitivo. Un ingresso binario specificato (anche se sconosciuto) in un cubo primitivo è invece un valore *richiesto*, quindi una *u* su un ingresso non permette di generare il corrispondente valore di uscita.

Si consideri ad esempio un AND a due ingressi, il cubo *u0*|*x* dà corrispondenza col cubo primitivo (il valore dell'uscita viene comunque valutato a 0), ma *u1*|*x* non la dà.

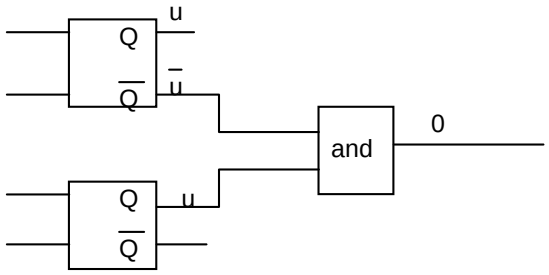
La logica a tre valori comporta una perdita d'informazione; per il NOT, dato che a una *u* sull'ingresso corrisponde una *u* sull'uscita, si perde proprio la relazione di complementazione. Come conseguenza, circuiti per i quali in realtà l'uscita potrebbe essere valutata verranno considerati a uscita incognita: si veda ad esempio la seguente semplice rete logica:



Figura

Il risultato della simulazione a tre valori è pessimista: in realtà, l'uscita del circuito è definita per qualunque valore dell'ingresso indefinito, e vale 1, ma le regole della logica a tre valori le assegnano valore u.

Effetti analoghi si presentano fra le due uscite di un flip flop, che non appaiono più complementari l'una all'altra. L'introduzione della *u* complementata in realtà non risolve il problema, salvo nel caso molto particolare in cui sia presente una sola variabile di stato; altrimenti, può addirittura portare a soluzioni errate (chiaramente da evitare molto più attentamente che quelle pessimistiche!). Ad esempio:



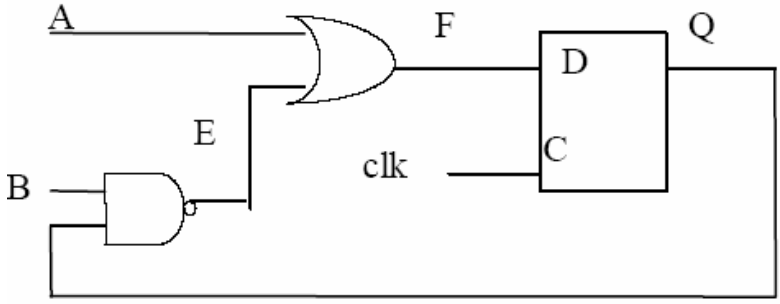
Figura

Una soluzione corretta implicherebbe l'uso di un segnale incognito u_i per ogni variabile di stato, con le regole $u_i \cdot \overline{u_i} = 0$ $u_i + \overline{u_i} = 1$; questa tecnica diventa però intrattabile per un numero elevato di variabili di stato, dato che per alcune linee il valore sarebbe rappresentato da lunghe espressioni logiche nelle variabili incognite.

Una volta definita l'estensione dell'algebra con cui operano i simulatori, si può fornire qualche ulteriore indicazione sui simulatori stessi.

Nella *simulazione compilata*, il modello, in codice compilato, diventa parte del simulatore: al limite, il simulatore ***è*** il modello in codice compilato. In questo ultimo caso, il codice ha anche il compito di leggere i vettori d'ingresso e produrre i vettori dei risultati; più in generale, il modello è collegato al nucleo del simulatore che ha fra i suoi compiti quelli di leggere i vettori degli ingressi, eseguire il modello per ogni vettore e fornire i risultati.

Si consideri il circuito sincrono in figura, controllato dal sincronismo CLK:



Figura

Si supponga che l'intervallo fra l'applicazione di un vettore d'ingresso e la successiva applicazione del clock sia sufficiente perché gli ingressi dei flip flop diventino stabili. In questo caso si possono ignorare i ritardi delle porte, e per ogni vettore d'ingresso basta calcolare in modo statico il valore di F e trasferire il valore su Q; è quindi possibile ricorrere alla simulazione compilata. Il modello in codice è generato in modo che il calcolo dei valori proceda livello per livello - ogni volta che il codice valuta una porta, le porte che ne forniscono

gli ingressi sono già state valutate. Per l'esempio considerato, la sequenza di istruzioni in uno pseudolinguaggio di descrizione sarebbe:

E:= NOT (B AND Q)

F:= A OR E

Q:=F

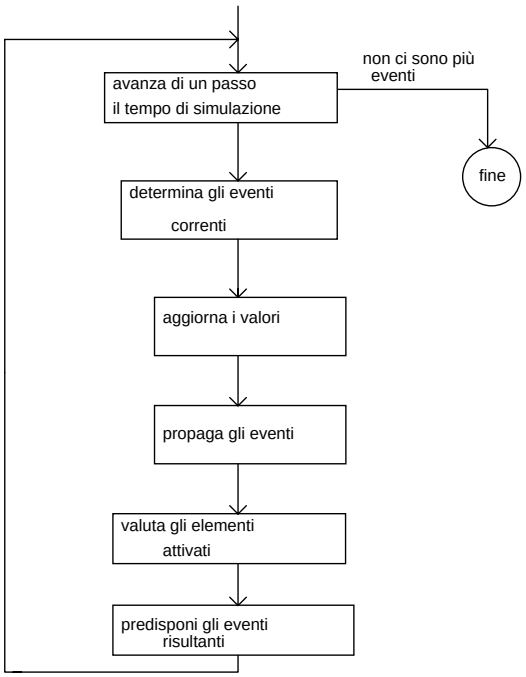
I valori degli ingressi primari A e B sono letti da un *file* d'ingresso. Se il valore iniziale di Q è noto, la simulazione è una normale simulazione binaria; altrimenti, il simulatore deve elaborare i valori 0,1,*u*,. Nel caso in cui il circuito da simulare sia combinatorio e si possa quindi usare semplicemente la logica a due valori, il valore di ogni segnale può essere rappresentato da un solo bit: si può allora pensare che - se a ogni linea di segnale è associata una parola di *K* bit - durante una passata di simulazione si elabori "in parallelo" il comportamento del circuito in risposta a *K* diverse configurazioni d'ingresso. Questo permette di ridurre i tempi di simulazione, quando questa debba essere ripetuta su un numero sufficientemente alto di configurazioni d'ingresso.

La simulazione compilata non è accurata per i circuiti asincroni (il funzionamento dipende dai valori di dati ritardi). A meno di creare modelli molto sofisticati - e il cui funzionamento dipende dalla calibrazione dei ritardi relativi e dal posizionamento dei ritardi concentrati equivalenti, lungo gli anelli di ritardo.- una simulazione statica come quella compilata non è in grado di trattare il fenomeno.

Nella *simulazione event-driven*, il simulatore usa un modello strutturale del circuito (quali i grafi visti in precedenza) per propagare gli eventi, e richiama opportune routines per valutare gli elementi componenti del circuito. Le variazioni dei valori degli ingressi primari sono definite nel *file* degli stimoli; gli eventi sulle linee "interne" vengono prodotti dalla valutazione degli elementi attivati.

Un evento si verifica in una dato istante di *tempo simulato*. Il *meccanismo di flusso del tempo* del simulatore manipola gli eventi in modo che si presentino nel corretto ordine temporale. Gli stimoli applicati sono rappresentati da sequenze di eventi che si verificano in istanti predefiniti; gli eventi che dovranno avvenire "in futuro" rispetto al tempo simulato attuale si dicono in attesa (*pending)* e vengono mantenuti in una *lista degli eventi*.

Si consideri il flusso concettuale di una simulazione event-directed:



Figura

Il tempo di simulazione viene fatto avanzare fino al prossimo istante per cui esiste un evento atteso; questo istante diventa il tempo di simulazione presente. Il simulatore preleva dalla lista degli eventi tutti gli eventi previsti per l'istante presente e aggiorna i valori dei segnali attivi. Si segue la lista di fanout dei segnali attivi per determinare gli elementi attivati (questo rispecchia la propagazione dei segnali nei circuiti reali).

La valutazione degli elementi attivati può a sua volta generare nuovi eventi: questi vengono programmati per il futuro in corrispondenza dei ritardi associati al funzionamento degli elementi, e il simulatore inserisce i nuovi eventi così generati nella lista degli eventi. La simulazione procede finché c'è attività logica nel circuito, cioè fino a quando la lista degli eventi non diventa vuota.

Dato che il tempo compare esplicitamente nella simulazione event-driven, occorre definire i *modelli di ritardo*. Ogni porta introduce un ritardo sui segnali che l'attraversano: il modello di una porta può essere costruito separando la funzione della porta dal ritardo (che viene schematizzato con un elemento di ritardo posto in cascata alla "porta ideale"). Si possono considerare due tipi di ritardi:

Ritardi di trasporto: costituiscono il modello fondamentale, che precisa il ritardo *d* fra la variazione dell'uscita e la variazione dell'ingresso /degli ingressi che l'hanno provocata. Normalmente, i ritardi vengono rappresentati come numeri interi: se tutti i ritardi in un circuito vengono considerati uguali, i ritardi vengono tutti normalizzati a 1 e si ha un *modello a ritardo unitario*

Ritardi inerziali: un circuito logico, per commutare, richiede energia. L'energia associata a un impulso di segnale è funzione di ampiezza e durata dell'impulso stesso: se la durata è troppo breve, il segnale non provocherà la commutazione della porta. La durata minima di un impulso

di segnale in ingresso necessaria per provocare la commutazione è il *ritardo inerziale (d)*: un impulso di durata inferiore a tale valore è uno "spike" che viene *filtrato* dalla porta.

Simulazioni temporali più complesse possono richiedere ulteriori informazioni (in particolare, in rapporto alla commutazione dei bistabili). Quando si opera su modelli di livello più elevato (RT o funzionale) si definiscono normalmente ritardi *di trasporto* associati alle varie funzioni eseguite.

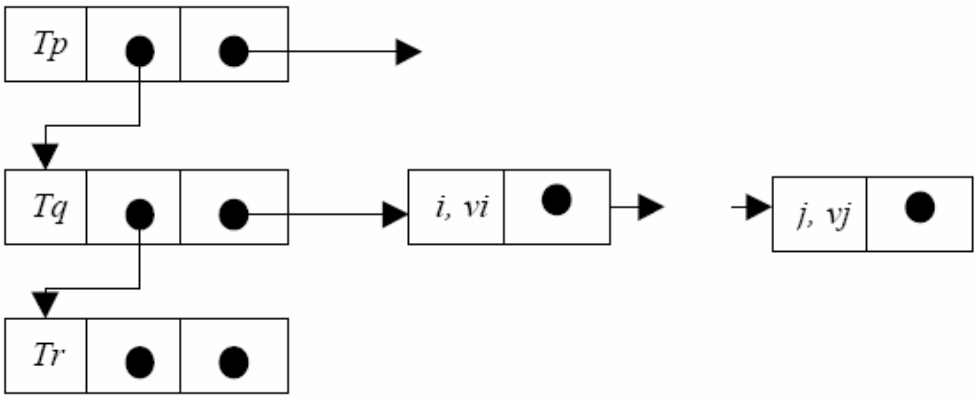
Sulla base dell'informazione strutturale, funzionale e - se necessario - temporale, il simulatore procede alla *valutazione degli elementi*.

La valutazione di un elemento combinatorio è il calcolo dei valori di uscita sulla base dei valori degli ingressi; la valutazione di un elemento sequenziale dipende anche dallo stato presente e produce, oltre all'uscita, anche lo stato prossimo.

Le tecniche di valutazione usate nei simulatori dipendono da molti fattori quali il sistema di valori logici usati nella simulazione, il modo di memorizzare i valori, il tipo degli elementi e il modo di modellarli. Si consideri ad esempio la valutazione di un elemento combinatorio G per una data configurazione dei suoi ingressi. In una prima soluzione, i valori dei segnali sono memorizzati in una tabella "parallela" alle tabelle dei segnali. Trovare i valori degli ingressi di G è un procedimento indiretto: la routine di valutazione deve inseguire le liste di fanin di G e poi accedere ai valori degli ingressi. In una seconda soluzione, si memorizzano i valori di ingresso di G in un'area di memoria associata a G stesso. Apparentemente, si spreca memoria (se un segnale ha un fanout a *k* rami, si replica il suo valore registrandolo nell'area di ognuno dei *k* elementi a cui i rami di fanout si collegano). I vantaggi sono:

- maggiore velocità delle routines di valutazione
- possibilità di "impaccare" insieme i valori
- adattabilità alla simulazione di guasto (in presenza di guasto, il valore sulla radice del fanout e quelli sui rami non sono necessariamente tutti identici).

Si consideri ora una possibile tecnica di simulazione *event-driven*: per semplicità, si ammette che tutte le porte introducano lo stesso ritardo nominale di trasporto. La *lista degli eventi* è in realtà costituita da un insieme di liste concatenate fra di loro; in ogni lista vengono memorizzati gli eventi che condividono lo stesso istante di attività, e due liste corrispondenti a due istanti successivi sono concatenate l'una all'altra, come in figura (dove $t_p<t_q<t_r$)



Figura

L'elemento (*i'*,*vi'*) nella lista associata a t_q indica che nell'istante t_q il segnale sulla linea *I* assumerà il valore *vi'*; nel corso della valutazione, nuovi valori possono essere inseriti nelle liste. Si noti che non sempre la valutazione di un componente e il successivo inserimento di un nuovo valore in una lista genera un evento (il valore del segnale può restare immutato): se si vuole che la simulazione, passando da una configurazione d'ingresso a quella successiva, comporti solo la valutazione degli elementi ai cui ingressi si verifica effettivamente un'attività, sarà necessario che l'algoritmo effettui una seconda passata per controllare che non si rivaluti un componente già valutato.

C a p i t o l o

IL PROBLEMA DELLA QUALITÀ

La **garanzia della qualità** è affidata sostanzialmente alla realizzazione di un *collaudo* soddisfacente, mediante la generazione di un insieme di configurazioni d'ingresso (*vettori di test*) da applicare, nella fase di collaudo, al dispositivo o al sistema, così da raggiungere una data *copertura* (rapporto fra il numero dei guasti *previsti* che i vettori di test consentono di rilevare e il numero totale di guasti previsti). Si deve sottolineare questo fatto: il collaudo - dato che la ricerca dei *difetti fisici* è praticamente impossibile - fa sempre riferimento a guasti di cui si precisano il tipo, la distribuzione spaziale (uno o più guasti presenti simultaneamente, distribuiti in modo casuale nel sistema oppure raggruppati perché interdipendenti, etc.), la distribuzione nel tempo (per il collaudo durante il tempo di vita del sistema, si precisa se i guasti si presentano uno per volta oppure simultaneamente) e altre caratteristiche. Il collaudo fa cioè riferimento a un *modello di guasto*, e la credibilità dei risultati è tanto migliore quanto più il modello di guasto adottato rispecchia la realtà fisica.

Il progetto iniziale - in genere - non offre *a priori* nessuna garanzia che il livello di copertura richiesto possa essere raggiunto. La soluzione più elementare, che garantisce la massima copertura possibile, consisterebbe nel ricorrere a tecniche esaustive (che utilizzino cioè *tutti* i possibili vettori di test): non è però accettabile per circuiti di complessità anche modesta, dato che richiederebbe tempo eccessivo per la *applicazione* del test. Si ricorre quindi a metodi per *generare* un insieme ridotto di vettori di test che garantiscano ugualmente copertura adeguata. Si vedranno nel seguito tecniche di generazione dei vettori di test su base *algoritmica*, introducendo i parametri di costo e prestazioni che permettono di valutarne i metodi.

Si rende opportuno, soprattutto al crescere della complessità dei dispositivi e sistemi sottoposti a collaudo, valutare a priori la "difficoltà" del collaudo stesso, cioè la difficoltà di identificare l'insieme di vettori di test necessari; si tratta di compiere sul sistema una *analisi di testabilità* (*testability analysis*), il cui risultato dovrebbe anche fornire al progettista qualche indicazione sulle aree del sistema che sarebbe opportuno modificare per rendere più facile il collaudo. L'*analisi della testabilità* diventa quindi un passo nella catena del progetto.

Intervenire su un sistema già completamente definito è una soluzione potenzialmente molto costosa; senza opportune tecniche di progetto, non c'è modo di garantire che le modifiche a un progetto difficile da collaudare lo migliorino effettivamente. Si introducono quindi metodologie di *progetto per la testabilità* (Design For Testability) grazie alle quali si può garantire una determinata semplicità di collaudo per il sistema che si vuole progettare. Inevitabilmente, queste tecniche rischiano di aumentare il costo del sistema (valutato come numero di porte logiche o come area di silicio) e ridurne le prestazioni temporali (cioè la velocità di funzionamento); occorre valutare l'*aumento di costo* del dispositivo/sistema finale come conseguenza dell'uso di queste tecniche di progetto a fronte del miglioramento in termini di costo del collaudo e di garanzia di qualità del prodotto.

Un passo ulteriore che mira a ridurre i costi di *applicazione* del collaudo (le macchine usate a questo scopo sono molto costose) consiste nel garantire che il dispositivo/sistema sia capace di *generare automaticamente* i vettori di test ed eventualmente di *valutare autonomamente* il risultato del test. Si tratta di introdurre tecniche di *Built-In Self Test*. Esistono soluzioni che offrono un

compromesso fra *complessità aggiuntiva* introdotta nel circuito (e quindi costo aggiuntivo) e *prestazioni ottenibili* in termini di copertura di guasto raggiungibile. Usandole, *non occorre una macchina che applichi i vettori di test e verifichi i risultati*: basta fornire al circuito un opportuno stimolo.

In questa sezione, si affronteranno nell'ordine i problemi citati.

Il problema del collaudo

Si tratta di uno dei più importanti nella vita di un sistema; può essere affrontato in momenti e con scopi diversi. Come prima cosa, si possono distinguere la attività di collaudo a seconda che vengano svolte *al termine della produzione* (prima cioè che il dispositivo o il sistema inizi il suo normale funzionamento) oppure *durante l'esercizio* del sistema finale che include il dispositivo o il sottosistema digitale di cui ci occupiamo.

1. Il test al termine della produzione è orientato essenzialmente alla ricerca dei *difetti di produzione*, che si manifestano o immediatamente o nelle prime ore di vita del sistema (molto spesso, nel caso dei dispositivi integrati, questi vengono sottoposti a una fase di *burn-in*, costituita da un insieme di prove di vita "accelerate" in condizioni di funzionamento estreme);
2. il test in esercizio (*run-time)* è orientato alla ricerca dei guasti che si manifestano durante la vita attiva del sistema, sia in conseguenza di difetti iniziali non manifestatisi prima, sia come conseguenza di usura e/o cattivo utilizzo.

Il test può essere rivolto:

a. al solo **rilevamento** dei guasti (*fault detection*): in questo caso, non è seguito da interventi di riparazione, ma il dispositivo o sistema riconosciuto guasto viene semplicemente scartato. Può essere un test del tipo "passa/non passa ("go/no go"): non si cerca di identificare *l'origine* del malfunzionamento, ma solo di accertare l'esistenza del malfunzionamento;

b. alla **localizzazione** dei guasti (*fault diagnosis*): molto più complesso (e quindi più costoso) del precedente, ha interesse se è possibile l'intervento di riparazione o se si vogliono realizzare statistiche sul processo di produzione (tipicamente, nella fase di messa a punto di tale processo).

A livello di dispositivo normalmente ci si accontenta del rilevamento (salvo rarissimi casi, non è possibile un intervento riparatore). A livello di sistema, si mira alla localizzazione del guasto all'interno della *più piccola parte economicamente sostituibile*.

***Cause di difetti e guasti* sono** ben diverse per dispositivi e sistemi, e dipendono fortemente dalla tecnologia di realizzazione. Difetti e guasti macroscopici a livello di sistema (interruzione di collegamenti, cortocircuiti...) vengono normalmente identificati con strumenti e tecniche diversi da quelli usati per i guasti a livello di dispositivo. Per tutti gli altri difetti e guasti, e per i malfunzionamenti che ne derivano, l'analisi può essere compiuta a diversi *livelli di astrazione*. Ai vari livelli a cui un circuito digitale può essere osservato, modellato e descritto corrispondono altrettanti livelli per *modellare i guasti* e *definire i test*.

A **livello fisico** si manifestano dei **difetti**. Tali sono ad esempio, nel caso di circuiti integrati, le perforazioni microscopiche nell'ossido di silicio, l'imperfetto allineamento fra maschere d'integrazione che porta a scomparsa o comparsa di giunzioni, i corti circuiti fra piste metalliche adiacenti dovuti a sbavature di metallo.

A **livello elettrico,** il difetto fisico *può* provocare una **malfunzione:** come conseguenza di un difetto, i valori di tensione e corrente misurati nei punti di test possono risultare diversi dal previsto, almeno per qualche configurazione di valori dei segnali d'ingresso.

A **livello logico**, il circuito viene descritto come interconnessione di porte logiche. Si distinguono *guasti che influenzano la funzione logica* e *guasti di ritardo* (*delay faults*) che influenzano la velocità di funzionamento del sistema o di qualche sua parte (in effetti, come conseguenza dei

ritardi può succedere che anche i valori delle funzioni risultino errati, almeno per qualche intervallo di tempo). Ci si occuperà prevalentemente della prima classe: come conseguenza di un *guasto logico*, il **valore logico del segnale su una linea** - in determinate condizioni - risulta diverso da quello previsto. I guasti definiti in relazione a un modello strutturale del circuito vengono definiti *guasti strutturali*.

A livello di astrazione più elevato, il circuito può essere descritto mediante la sua **funzionalità**, prescindendo dalla realizzazione interna: ad esempio, una macchina a stati finiti può essere descritta mediante la tabella degli stati, un circuito aritmetico mediante la funzione svolta. I guasti logici possono generare **errori funzionali** (la funzione richiesta non viene svolta correttamente, o ne vengono svolte altre non volute). Esistono in genere più livelli funzionali possibili (si pensi a una rete combinatoria realizzata come rete di multiplexer: si può fare riferimento agli errori funzionali definiti a livello dei multiplexer, o a quelli dell'intera funzione).

Per dispositivi di grande complessità funzionale (tipicamente, i microprocessori), in cui uno stimolo provoca un concatenarsi di funzioni, si può prendere in considerazione il livello *comportamentale* (behavioral). Nel caso di un microprocessore, si identifica il *comportamento* del dispositivo quando esso deve eseguire una determinata istruzione. In corrispondenza, si definiscono gli *errori comportamentali*. Anche per il comportamento è possibile considerare più livelli di astrazione: in termini comportamentali si opera tipicamente nel collaudo (sia in produzione, sia in esercizio) di sistemi o sottosistemi di elaborazione (eventualmente inseriti in applicazioni dedicate), quando si analizza il comportamento dell'unità sotto test *alle sue interfacce* (ai "piedini" del microprocessore, sul bus di collegamento della piastra...). Si possono definire anche diverse *granularità temporali* (si analizza il comportamento a ogni singolo ciclo di clock, a ogni accesso ai dati, in corrispondenza di particolari segnali di controllo...).

Nella prima classificazione operata finora, si sono presi in considerazione i guasti di tipo *permanente*: guasti cioè che - una volta insorti - continuano a presentarsi, ovviamente a pari condizioni (per la stessa configurazione degli ingressi nel caso di reti combinatorie, per la stessa configurazione stato presente/ingressi primari nel caso di reti sequenziali). Particolarmente durante l'esercizio, si verificano in realtà anche guasti *transitori* (che scompaiono dopo un breve intervallo di tempo) e guasti *intermittenti* (che scompaiono e ricompaiono nel tempo); per studiarli occorrono dati statistici raramente disponibili e comunque non facilmente estrapolabili da un esempio a una generalità di casi. Per questo motivo, gli studi (e lo sviluppo di strumenti CAD) si sono concentrati quasi esclusivamente sui *guasti permanenti*, che saranno gli unici trattati in questo corso.

In pratica, in ambito digitale fra i vari livelli di astrazione sopra citati i primi due (fisico ed elettrico) non vengono trattati, se non in casi molto particolari. Si opera a livello fisico solo per *caratterizzare un processo produttivo* (è necessario ricorrere a tecniche estremamente lente e costose). Anche l'analisi a livello elettrico è molto complessa e di difficile valutazione (di fatto, il test di circuiti analogici è un problema di grande complessità e tutt'altro che risolto); si adotta l'analisi a livello elettrico solo per circuiti CMOS su modelli fortemente semplificati (switch-level). Si discuteranno quindi in dettaglio i modelli più usati - logico, funzionale - oltre a modelli specifici che ben caratterizzano alcune particolari famiglie di componenti.

Il **modello di guasto logico** è il modello più ampiamente usato fino ad oggi: fu introdotto da Poage nel 1963 per componenti bipolari, ma si è dimostrato sperimentalmente accettabile

anche col cambiare delle tecnologie, in particolare (sia pure con alcune restrizioni e avvertenze) per i circuiti MOS.

In questo modello, i guasti vengono *riferiti alle linee di interconnessione* (o "linee di segnale" - in inglese, *nets*) fra i componenti logici: si noti che questo è - appunto - un *modello* che schematizza la *conseguenza* di un guasto fisico - nella realtà, *fisicamente* il difetto che provoca il guasto danneggia un componente e lo rende incapace di funzionare correttamente. Si suppone che, in conseguenza del guasto che le viene associato, una linea resti *bloccata al valore logico 0 (stuck-at-0*, in genere abbreviato come *s-a-0) o al valore logico 1 (stuck-at-1, s-a-1)*, indipendentemente dal valore logico corretto generato dalla porta che alimenta la linea stessa. Occorre sottolineare che *questo modello* ***non*** *i veri guasti delle linee di interconnessione - circuiti aperti o cortocircuiti - ma guasti delle giunzioni su cui la linea inizia o termina.* I guasti fisici delle interconnessioni - in particolare, la presenza di un cortocircuito fra due linee adiacenti - richiedono modelli molto più complessi (se non addirittura l'uso di particolari tecniche di microscopia elettronica) e fortemente legati alla particolare tecnologia implementativa: si è però verificato sperimentalmente che la ricerca dei guasti logici riesce a mettere in evidenza anche una buona percentuale di guasti fisici sulle linee.

In teoria, un modello completo di guasto logico imporrebbe di considerare *tutte le possibili combinazioni di guasti stuck-at*. In un circuito con n linee, però, questo porterebbe a considerare $3^n - 1$ possibili combinazioni di guasti (ogni linea può essere sana, oppure *s-a-0* o ancora *s-a-1*, e si devono esaminare tutti i casi all'infuori di quello in cui tutte le linee sono sane). Chiaramente, per un numero anche molto ridotto di linee interne questo porterebbe a un numero di casi assolutamente intrattabile. Già fin dai primi lavori mirati alla ricerca dei guasti si è introdotta una ipotesi fortemente semplificatrice, e cioè quella di **guasto singolo**: si suppone cioè che *nel sistema sia presente al più un guasto logico*. La complessità della ricerca dei guasti diminuisce drasticamente: il circuito con n linee presenta infatti *2n guasti singoli*. Evidentemente, ci si chiede quanto questa ipotesi sia giustificata: se era accettabile per circuiti semplici e realizzati con componenti discreti come quelli cui si riferiva Poage, certo non lo è per il collaudo in produzione dei moderni circuiti integrati, nei quali possono essere presenti simultaneamente più guasti o comunque *un singolo difetto fisico può produrre guasti logici multipli*. Può essere più accettabile per il collaudo in esercizio, se supportata da una *strategia di collaudo frequente* (il collaudo viene ripetuto con frequenza atta a garantire che la probabilità che si presentino più guasti nell'intervallo fra due collaudi sia trascurabile), ma anche in questo caso, se un'azione di collaudo non rileva *tutti* i guasti singoli possibili (non dà copertura massima), l'azione successiva dovrebbe comunque affrontare guasti multipli. Si è però sperimentalmente rilevato che ***nella maggior parte dei casi, guasti multipli possono essere rilevati dalle azioni di collaudo progettate per i vari guasti singoli che contribuiscono a formare il guasto multiplo***, e che se la copertura dei guasti singoli è molto elevata (superiore al 95%) anche la copertura dei guasti multipli è accettabile: di conseguenza, l'ipotesi di guasto singolo viene generalmente adottata.

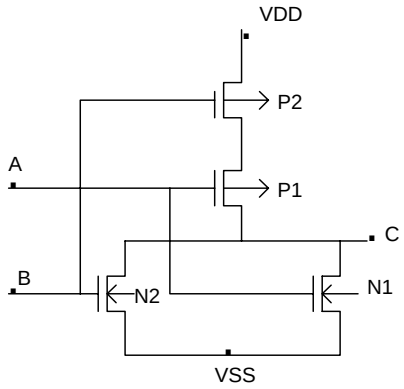
Si considerano ora alcuni altri modelli di guasto particolarmente interessanti: Anche per questi casi (all'infuori del caso delle memorie RAM) si fa riferimento di solito a ipotesi di guasto (o di errore, a livello funzionale) singolo.

Guasti a livello di transistore.

Il modello ha interesse particolarmente per circuiti C-MOS. Il transistore MOS è considerato come un interruttore ideale (*switch*); i guasti possibili sono:

1. transistore permanentemente bloccato in modo da costituire un circuito aperto (*stuck-at-open*);
2. transistore permanentemente bloccato in modo da costituire un corto circuito (*stuck-at-on*);

Per mettere in evidenza le conseguenze di questi guasti, si consideri una porta NOR in tecnologia C-MOS:



P1 (P2) conduce quando l'ingresso A (B) vale 0; al tempo stesso N1 (N2) è aperto (quindi non conduce). Di conseguenza, per A = B = 0 è C=V_{DD} (C è isolato da V_{SS}) mentre, se A o B (o ambedue) valgono 1, C viene collegato a V_{SS} e isolato da V_{DD}.

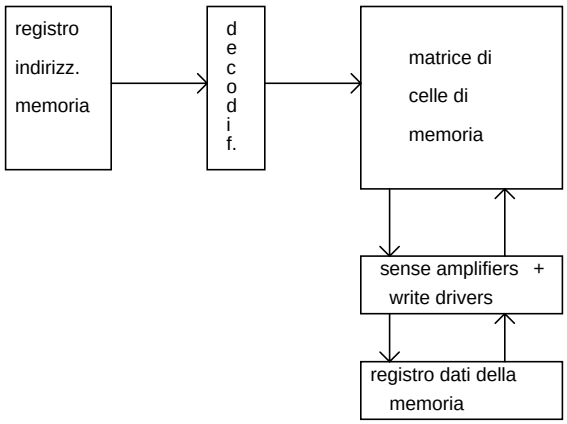
Si supponga che il transistore P1 sia stuck-at-open: C non risulta mai collegato né a V_{SS} né a V_{DD}: è in stato di *alta impedenza* (floating). Se C è collegato a un carico capacitivo (es., un'altra porta CMOS) il valore di tensione su C viene mantenuto per un periodo di tempo anche molto lungo (dipendente dalle correnti di dispersione del condensatore): si manifesta cioè un *fenomeno di memoria* in un circuito intrinsecamente combinatorio. Perché il circuito *produca errore* quando si applica il vettore di test 00, occorre prima portare C a 0: il guasto quindi viene rivelato dalla *coppia di vettori* 10 (porta C a 1) e 00 - mentre per il guasto *A s-a-1* sarebbe bastato il solo vettore 00. In genere, i guasti *s-a-open* richiedono *una successione di due vettori di test per essere rivelati.*

Anche i guasti di tipo *stuck-at-on* creano problemi analoghi. Un insieme di test progettato per guasti logici non copre quindi tutti i guasti presenti (a livello di transistori) in un circuito CMOS. Per poter utilizzare il modello logico si renderebbe necessario sostituire la porta CMOS con un circuito logico equivalente più complesso, in cui si inserisca un blocco che congloba l'effetto di memoria. Vale la pena di dire che in realtà molti dispositivi C-MOS vengono collaudati esclusivamente con test progettati per i guasti logici; fra l'altro, al diminuire delle geometrie (cioè al crescere della miniaturizzazione) tale modello sembra dare crescente soddisfazione.

Modelli di guasto per circuiti particolari: le memorie RAM.

Le caratteristiche strutturali delle memorie (più densamente impaccate della logica, sui chip) rendono opportuno l'uso di particolari modelli di guasto (e corrispondenti tecniche di test): si ricorre quindi a particolari modelli**.** I tipi di guasti considerati sono:

1. Guasti *parametrici*: si tratta, ad esempio, di livelli d'uscita inaccettabili, consumo troppo elevato, margini di rumore inaccettabili, etc. Hanno natura legata alla tecnologia e al processo produttivo;
2. Guasti *funzionali*: sono legati a uno *schema funzionale* della memoria:



In teoria, anche escludendo i guasti parametrici sarebbe necessario controllare il buon funzionamento di ogni blocco funzionale e di ogni cella della memoria *per tutte le possibili combinazioni di valori registrati nelle altre celle*. Si dovrebbero cioè esaminare:

- I guasti stuck-at nelle celle di memoria, nella logica di decodifica, nei registri di indirizzamento e di dati;
- I guasti di accoppiamento (*coupling faults*): due celle sono accoppiate se il cambiamento di stato dell'una provoca variazione di stato dell'altra.
- I guasti *pattern sensitive*, che alterano il contenuto di una specifica cella quando nelle altre celle si presenta una particolare distribuzione di 0 e 1, o che hanno come conseguenza l'impossibilità di leggere o scrivere in una data cella quando nelle altre celle si presenta una particolare distribuzione di 0 e 1.

Di fatto, anche per le RAM si ricorre a modelli fortemente semplificati per cui sono stati definiti metodi di test che (sperimentalmente) consentono di rilevare un gran numero dei vari guasti sopra citati con soluzioni economicamente accettabili.

Modelli di guasto in dispositivi particolari: le PLA

Le PLA (*P*rogrammable *L*ogic *A*rrays) sono strutture logiche molto dense e di implementazione efficiente su silicio; se all'inizio degli anni '80 venivano proposte come componenti "semi-custom" a sé stanti, mediante i quali l'utente poteva realizzare arbitrarie funzioni logiche, oggi nel campo dei semi-custom sono state quasi totalmente rimpiazzate da strutture quali "gate arrays", ma vengono invece usate molto spesso per realizzare parti di circuiti integrati di grandi dimensioni (un uso tipico riguarda l'unità di controllo di microprocessori o microcontrollori). Data la struttura regolare delle PLA, che dà un'immediata corrispondenza fra schema logico e disegno fisico (cosa che non vale per i circuiti "in logica sparsa"), e data la grande densità d'impaccamento del disegno fisico sui chip, si considerano guasti particolari derivanti *dal disegno fisico* (layout).

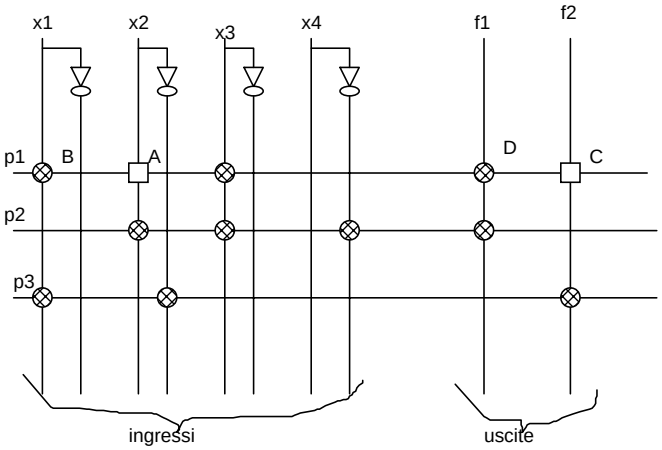
Si ricorda brevemente come la PLA contenga una *sezione AND* in cui le *linee d'ingresso* incrociano le *linee di parola* (o *di prodotto*) e una *sezione OR* in cui le linee di parola incrociano le *linee di uscita* o *linee di bit* (o *di funzione*). Quando una variabile (naturale o complementata) compare in un *prodotto*, all'*incrocio (crosspoint)* fra le corrispondenti linee deve esistere un dispositivo che le collega (tipicamente, un transistore): lo stesso vale quando un prodotto deve comparire fra i termini di un OR.

Si consideri ad esempio una PLA che realizzi le due funzioni:

$f_1 = p_1 + p_2 = x_1x_3 + x_2x_3\overline{x_4}$

$f_2 = p_3 = x_1\overline{x_2}$

(sul disegno, i cerchietti indicano i collegamenti fra le linee: si prescinde dalla particolare tecnologia, che condurrebbe a distinguere soluzioni NOR-NOR piuttosto che NAND-NAND).



La mappa di Karnaugh associata alla PLA è la seguente:

x3x4 \ x1x2	00	01	11	10
00				01
01				01
11			10	10
10		10	10	10

(con la doppia riga sottile si sono indicati i due implicanti di f_1, con la doppia riga spessa l'implicante di *f2*) I guasti che si considerano nel caso della PLA sono:

1. Guasti stuck-at (0,1) su ingressi, uscite, negatori di ingresso, linee di prodotto;
2. Guasti d'incrocio (*crosspoint faults*): un guasto singolo d'incrocio corrisponde all'assenza o all'aggiunta di un singolo dispositivo nella matrice AND o OR;

Si è già parlato dei guasti s-a; si considerano ora in dettaglio gli effetti dei **guasti d'incrocio:**

a) presenza di un dispositivo addizionale nella matrice AND (caso **A** nel disegno): il corrispondente prodotto viene modificato dall'*aggiunta* della variabile sulla linea d'ingresso - l'implicante diventa *più piccolo*, cioè corrisponde a un sottocubo più piccolo sulla mappa di Karnaugh. Il guasto si dice ***shrinkage fault***. Se gli 1 lasciati "scoperti" dalla riduzione del sottocubo non sono coperti da nessun altro implicante per almeno una delle funzioni, qualsiasi mintermine corrispondente a tali 1 costituisce un test per il guasto;

b) presenza di un dispositivo addizionale nella matrice OR (caso **C** nell'esempio). La corrispondente funzione d'uscita contiene un termine prodotto in più. È un *appearance fault*. Qualsiasi mintermine coperto dal prodotto aggiunto e non compreso nella funzione corretta costituisce un test per il guasto.

c) mancanza di un dispositivo nella matrice AND (es. assenza di **B** nell'esempio dato): il corrispondente implicante diventa "più grande" (il sottocubo raddoppia le proprie dimensioni). È un *growth fault*: qualsiasi mintermine nel sottocubo espanso che non facesse parte della funzione nominale costituisce un test;

d) mancanza di un dispositivo nella matrice OR (es. **D** nella PLA data): *scompare* un termine prodotto da una funzione (*disappearance fault*). Qualsiasi mintermine presente in questo prodotto e non coperto altrimenti nella stessa funzione costituisce un test.

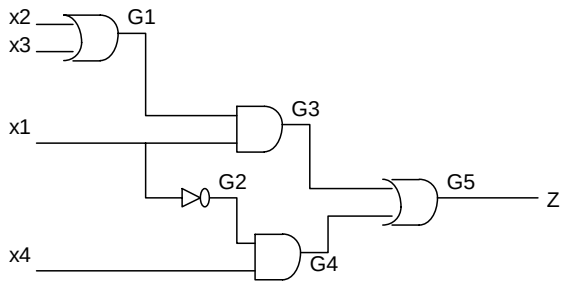
Un *cortocircuito* forza valori identici sulle due linee coinvolte: lo si modella con un OR o con un AND a seconda della tecnologia usata. Si è dimostrato sperimentalmente che un test che rilevi *i soli guasti d'incrocio* in realtà copre un gran numero di *tutti* i guasti.

Una volta scelto il modello di guasto, e definito quindi l'insieme di guasti che si vogliono rilevare (o eventualmente diagnosticare), occorre identificare un insieme di vettori di test che consentano di raggiungere la *copertura* voluta con il *costo* minimo; il concetto di "costo" richiederebbe in realtà qualche ulteriore precisazione - come si vedrà più dettagliatamente nel seguito, si può far riferimento al costo di *generazione* dei vettori di test (valutato in termini di complessità computazionale) o a quello di *applicazione* dei vettori stessi (di solito indicato sommariamente dal numero di tali vettori, cioé dalla "lunghezza" dell'esperimento di test).

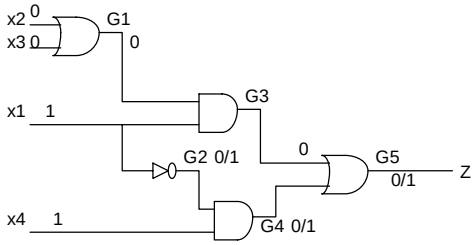
La generazione dei vettori di test

Si considererà inizialmente il test a *livello logico* delle *reti combinatorie* - di gran lunga più semplice del test delle reti sequenziali, e comunque indispensabile anche per quest'ultimo. Sia $Z(x)$ una funzione logica realizzata da un circuito combinatorio N, dove con x si indica un generico vettore di ingresso. (Con t si indicherà invece uno *specifico* vettore di ingresso, e con $Z(t)$ la risposta del circuito a tale vettore). Un guasto f trasforma il circuito N in un diverso circuito N_f (si suppone, ancora combinatorio) la cui risposta a un generico vettore d'ingresso è $Z_f(x)$. Il collaudo di N si effettua applicando una sequenza T di vettori di test $t_1,...,t_m$ e confrontando i risultati ottenuti con quelli previsti da parte di N (in una rete combinatoria, i vettori nella sequenza T possono essere applicati in ordine arbitrario). Un vettore di test t rileva il guasto f se e solo se $Z_f(t) \neq Z(t)$. In un circuito a una sola uscita, un test t che rileva un guasto f rende $Z(t)=0$ e $Z_f(t)=1$ o viceversa; l'insieme di tutti i guasti che rilevano f è quindi dato dalle soluzioni dell'equazione $Z(x) \oplus Z_f(x) = 1$. In teoria almeno, si potrebbe trovare un insieme di vettori di test che garantiscano la massima copertura consentita dal circuito risolvendo l'equazione di cui sopra per tutti i guasti compresi nel modello adottato (in pratica, una soluzione di questo genere ha complessità computazionale inaccettabile e comunque dà, solitamente, un insieme di vettori di test *ridondante* - uno o più guasti risultano coperti da più vettori di test, dato che in genere un singolo vettore di test permette di rilevare più guasti).

Si veda ad esempio il seguente circuito, che realizza $Z = (x_2 + x_3)x_1 + \bar{x}_1 x_4$ Sia $f=x_4$ *s-a-0*. In presenza del guasto la funzione diventa $Z_f = (x_2 + x_3)x_1$; è quindi $Z(x) \oplus Z_f(x) = \bar{x}_1 x_4$. Qualunque vettore di test nel quale sia $x_1=0$, $x_4=1$ costituisce un test per f.



Se ora si esamina il guasto *G2 s-a-1*, si vede che $Z(x) \oplus Z_f(x) = x_1 \overline{x_2}\,\overline{x_3} x_4$ - l'unico vettore di test per questo guasto è quindi il vettore *1001*. In effetti, si simuli il circuito dell'esempio sano, applicando tale vettore, e si simuli anche il circuito in cui sia stato "forzato" (*iniettato*) il guasto in questione: i risultati delle due simulazioni sono mostrati in figura (quando sono diversi per il circuito guasto e per quello sano, i due valori vengono indicati come v/v_f).



Il guasto - come previsto - viene rilevato, dato che il valore di uscita è diverso nei due casi.

Si identificano dall'esempio due concetti fondamentali nel rilevamento guasti:

1. un test *t* che rileva un guasto *f attiva f*, cioè genera un errore (un effetto di guasto) creando valori diversi di *v* e v_f nel punto di guasto.
2. *t propaga* l'errore a un'uscita primaria *w,* cioè fa si che tutte le linee lungo almeno un percorso dal punto di guasto a *w* abbiano diversi valori di *v* e v_f - nell'esempio, l'errore si propaga lungo il percorso *G2-G4-G5*. Una linea il cui valore nel test *t* cambia in presenza del guasto *f* è *sensibilizzata al guasto f dal test t*. Un percorso fatto di linee sensibilizzate è detto percorso sensibilizzato (sensitized path).

Una porta la cui uscita è sensibilizzata a *f* ha anche almeno un ingresso sensibilizzato a *f*. Data una porta logica, si dice che uno dei suoi ingressi ha *valore controllante c* se tale valore determina quello dell'uscita *indipendentemente dai valori agli altri ingressi*; si indica poi con *inversione i* della porta il valore tale per cui - ponendo a *c* uno degli ingressi - l'uscita della porta stessa vale $c \oplus i$. Nella seguente tabella si indicano I calori controllanti e di inversione per le principali porte logiche:

	c	*i*
AND	0	0
OR	1	0
NAND	0	1
NOR	1	1

Perché una porta possa propagare il percorso sensibilizzato, occorre che:

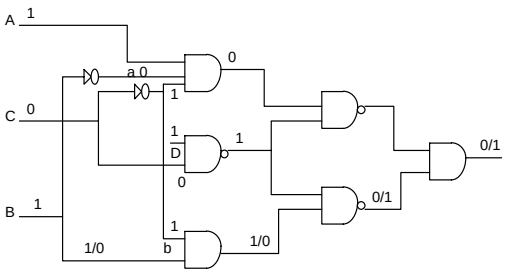
a) tutti gli ingressi di *G* sensibilizzati a *f* abbiano lo stesso valore *a*;

b) tutti gli (eventuali) ingressi non sensibilizzati a *f* abbiano valore $\overline{c}$;

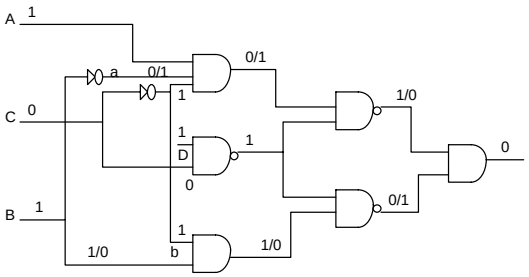
In questo caso, l'uscita di *G* ha valore $a \oplus i$.

Si dice che un guasto *f* è rilevabile se esiste un test *t* che rileva *f*; altrimenti, *f* è un guasto *non rilevabile*. Apparentemente, i guasti non rilevabili non creano problemi (anche se tali guasti sono presenti, il circuito continua a comportarsi correttamente); un guasto non rilevabile può però invalidare l'ipotesi di guasto singolo. Quando si genera un insieme di vettori di test, si mira a creare un *insieme di test che consenta il rilevamento completo*; se ci sono guasti non rilevabili, l'insieme di test può essere insufficiente a rilevare tutti i guasti rilevabili (perché i guasti non rilevabili possono presentarsi simultaneamente a un guasto rilevabile e quindi generare guasti multipli).

Si consideri ad esempio il seguente circuito:



si verifica che il guasto *b s-a-0* è rilevato da *t=1101*. Il circuito contiene anche un guasto non rilevabile - precisamente, *a s-a-1*: si può verificare che - se questo guasto è presente - *t* non è più in grado di rilevare *b s-a-0*:



Un circuito combinatorio che contiene guasti *s-a* non rilevabili è **ridondante**: lo si può sempre semplificare eliminando almeno una porta o un ingresso di una porta. Ad esempio, si consideri una porta AND G con un ingresso il cui guasto *s-a-1* sia non rilevabile: la funzione non cambia in presenza di questo guasto - quindi si può porre un 1 (invece di una variabile) sull'ingresso interessato. Ma un AND a *n* ingressi dei quali uno è fisso a 1 equivale a un AND con *n-1* ingressi ottenuto rimuovendo l'ingresso il cui valore è fisso a 1. Se invece un AND ha un ingresso il cui guasto *s-a-0* non sia rilevabile - quindi il cui valore è *sempre* 0, che è il valore controllante per l'AND - si può rimuovere l'AND e sostituirlo con un segnale fisso a 0. Le regole di semplificazione sono elencate qui sotto:

Guasto non rilevabile	regola di semplificazione
ingresso di AND (NAND) *s-a-1*	rimuovi l'ingresso
ingresso di AND (NAND) *s-a-0*	rimuovi la porta, sostituisci con 0(1)
ingresso di OR(NOR) *s-a-0*	rimuovi l'ingresso
ingresso di OR (NOR) *s-a-1*	rimuovi la porta, sostituisci con 1(0)

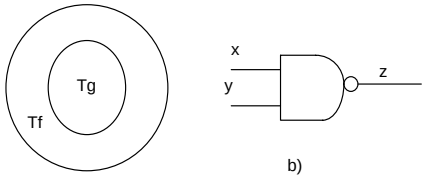
Un circuito combinatorio in cui tutti i guasti *s-a* sono rilevabili è *irridondante*. Si noti che la ridondanza può essere voluta, per garantire che il circuito sopravviva a determinati guasti mantenendo la propria funzionalità - si entra allora nel campo della tolleranza ai guasti, per cui esistono regole di progetto specifiche, e in cui occorre comunque garantire non solo la rilevabilità, ma addirittura la diagnosticabilità dei guasti a livello dei moduli per cui si chiede la tolleranza ai guasti. In ogni altro caso, è chiaro che riuscire a valutare la presenza di guasti

ridondanti (se non a rimuoverli) è fondamentale per potere ragionevolmente affermare che un certo esperimento di test ha raggiunto la massima copertura *possibile* per il circuito assegnato.

Anche una volta rimossi i guasti ridondanti, l'insieme dei *2n* guasti logici presenti in un circuito con *n* linee può essere eccessivo per la ricerca dell'insieme ottimo di vettori di test; in genere, però, esistono guasti fra loro *equivalenti*. Due guasti *f* e *g* sono equivalenti se i loro effetti non sono distinguibili alle uscite del circuito (è $Z_f = Z_g$, qualunque sia il vettore d'ingresso). La relazione di equivalenza induce una *partizione di equivalenza* sull'insieme dei guasti: ogni classe di equivalenza comprende tutti (e soli) i guasti che sono mutuamente equivalenti, mentre due guasti inseriti in due classi diverse *non* sono equivalenti. Nessun test è in grado di distinguere fra due guasti appartenenti a una stessa classe di equivalenza: anche quando l'esperimento mira alla *identificazione* dei guasti, in realtà si riesce solo a identificare la *classe di equivalenza* a cui il guasto appartiene. Per compiere una ricerca dei guasti basta quindi *considerare un solo guasto rappresentativo per ogni classe*. Questa scelta costituisce il cosiddetto *equivalence fault collapsing*, e costituisce (o può costituire) una fase preliminare nella definizione dell'insieme di vettori di test. Di fatto, operare un fault collapsing completo su un circuito è un problema *NP*-completo, quindi computazionalmente intrattabile: normalmente ci si limita a operare fault collapsing su singole porte logiche o su brevi percorsi privi di fan-out riconvergenti.

Per una porta AND a *n* ingressi, i guasti *s-a-0* su un qualsiasi ingresso e sull'uscita costituiscono un'unica classe di equivalenza, come per un NAND gli *s-a-0* su un qualsiasi ingresso e lo *s-a-1* sull'uscita, etc. Mentre a una porta a *n* ingressi si dovrebbero associare *2(n+1)* guasti di tipo *s-a*, le classi di equivalenza permettono di ridurli a *n+2* (per un AND, occorrono una classe che comprende tutti gli *s-a-0*, *n* classi che contengono ognuna un ingresso *s-a-1*, e una relativa allo *s-a-1* sull'uscita).

Se lo scopo è solo il *rilevamento* dei guasti, si può introdurre un ulteriore tipo di fault collapsing, detto *dominance fault collapsing*. Sia T_g l'insieme di tutti i test che rilevano un dato guasto *g*: si dice che un guasto *f* domina *g* se e solo se *f* e *g* sono *funzionalmente equivalenti rispetto a* T_g - in altre parole, se qualsiasi test che rileva *g* rileva anche *f*, ma non viceversa. Per *rilevare* (senza localizzare) i guasti è dunque inutile considerare il guasto *dominante f*, dato che trovando un test che rilevi *g* si rileva automaticamente anche *f*. (Val la pena di notare che alcuni autori invertono la definizione di dominanza - considerano cioè *g* dominante su *f*). Il concetto di dominanza può avere una rappresentazione grafica (v. fig. A); per esemplificarlo, si consideri ancora una porta NAND (fig. b). Siano il guasto *g* uno *s-a-1* sulla linea *y*, il guasto *f* uno *s-a-0* sulla linea *z*. Il guasto *g* può essere rilevato dall'unico test $T_g = 10$, che rileva anche *f*. Quest'ultimo però è rilevato anche dai vettori 01 e 00, quindi domina *g*.



a) b)

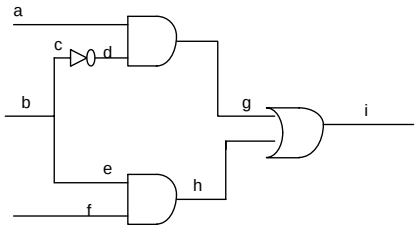
Più in genere, data una porta con valore controllante *c* e inversione *i*, il guasto d'uscita $s-a-(\bar{c} \oplus i)$ domina qualsiasi guasto d'ingresso $s-a-\bar{c}$: il guasto sull'uscita può allora essere tolto dall'insieme dei guasti per cui si cerca un test. Nel caso del NAND, si cercheranno vettori di test solo per la classe di equivalenza (*x:s-a-0, y:s-a-0,z:s-a-1*) e per i guasti (*x:s-a-1*) e (*y:s-a-1*).

Si può ripetere il fault collapsing partendo dagli ingressi primari finché non si raggiunge un *checkpoint*, cioè

- un ingresso primario, oppure
- un punto di diramazione di un fan-out.

Le classi di equivalenza che si ottengono non sono necessariamente massime, ma la procedura per determinarle non comporta "backtracking" ed è quindi computazionalmente trattabile. È stato inoltre dimostrato che, se tutti i guasti dei checkpoint sono rilevabili, è sufficiente determinare un test per tutti e soli i guasti nei checkpoint per garantire la copertura di *tutti i guasti s-a singoli nel circuito*.

L'identificazione delle classi di guasto sopra citata si basa su una semplice *analisi locale* del circuito. Si consideri il seguente esempio:



Si sono contrassegnate con lettere dell'alfabeto tutte le 9 linee del circuito , cui sono associati i guasti stuck-at: si noti che *radice e rami di un fan-out sono linee distinte* (fra i cui guasti esiste una relazione di dominanza ma non di equivalenza). I guasti singoli sono 18. Le classi di equivalenza vengono "fatte crescere" a partire dagli ingressi:

1. *a:s-a-0; c:s-a-1; d:s-a-0; g:s-a-0;*
2. *a: s-a-1*
3. *c: s-a-0; d:s-a-1;*
4. *b: s-a-0*
5. *b:s-a-1*
6. *e: s-a-0; f:s-a-0; h.s-a-0;*
7. *e: s-a-1*
8. *f: s-a-1*
9. *g:s-a-1; h:s-a-1;i s-a-1*
10. *i s-a-0.*

Le classi di equivalenza si riducono a 10.

La ricerca dei vettori di test si fa dunque con riferimento alle classi (di equivalenza o di dominanza) determinate mediante fault collapsing. Come già accennato, si potrebbero identificare con tecniche algebriche dei vettori di test per tutte le classi di guasto - ma si è già detto che tale soluzione risulta in realtà inapplicabile. Esistono anche metodi esatti che permettono di trovare un insieme minimo di vettori di test che coprano tutti i guasti non ridondanti - ma anche questi in pratica sono troppo computazionalmente onerosi per essere applicati. In genere, si ricorre a tecniche euristiche (se non addirittura casuali) per la generazione dei vettori di test; nelle soluzioni algoritmiche, si sceglie di volta in volta un guasto rappresentativo di una classe non ancora coperta e si cerca un vettore di test capace di rilevarlo. Dato che tale ricerca è anch'essa di notevole complessità, ovviamente si vuole evitare

la ricerca di vettori ridondanti - si vuole quindi stabilire quali altri guasti siano già coperti dai vettori trovati, in modo da ridurre lo spazio di ricerca. A questo scopo, si ricorre alla:

Simulazione di guasto

che consiste nella simulazione del circuito in presenza di guasti.

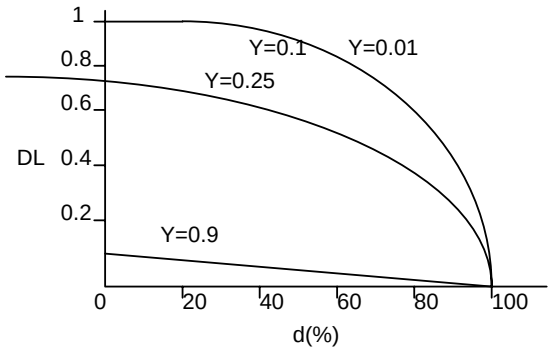
Applicando uno stesso insieme di test T in simulazione al circuito sano e al circuito guasto, si può stabilire quali guasti siano coperti da T. La simulazione di guasto può essere usata per **valutare un insieme di test T**: si identifica la copertura di guasto offerta da T. (La copertura di guasto non coincide con la copertura dei difetti fisici, ma l'esperienza ha dimostrato che c'è una buona relazione statistica).

Dalla qualità del test dipende la qualità del prodotto finito.

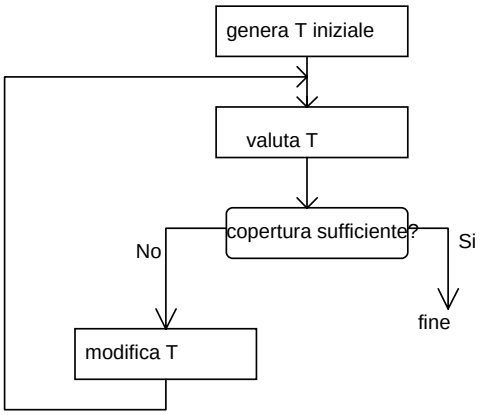
Siano *Y* la *resa di produzione* (cioè la probabilità che il dispositivo prodotto sia privo di guasti), *DL* il *livello di difettosità* (cioè la probabilità di vendere un prodotto difettoso), con *d* la *copertura di difetto* garantita dal test, la relazione fra resa e livello di difettosità è

$$DL=1-Y^{1-d}$$

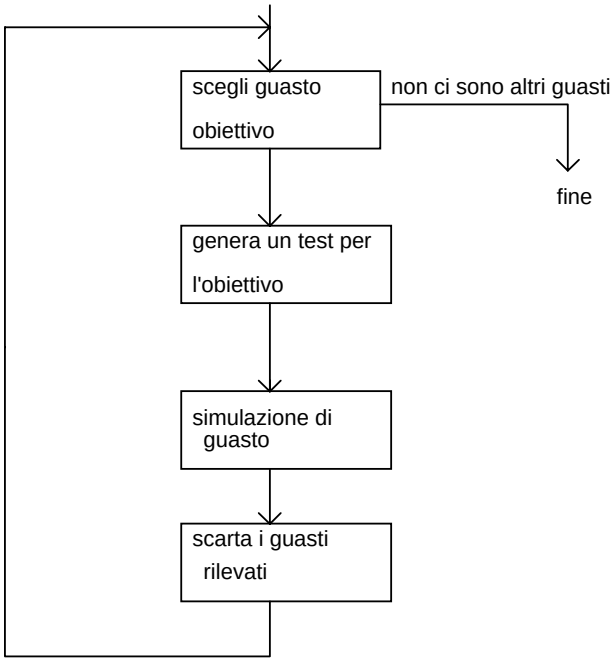
Se si accetta che la copertura di difetto sia approssimata dalla copertura di guasto, si può valutare la copertura di guasto necessaria per raggiungere un dato livello di difettosità: ad esempio, per un dato processo sia *Y=0.5*. Per raggiungere *DL=0.01* (si accetta cioè che 1% dei prodotti venduti siano difettosi!) occorre una copertura di guasto del 99% - in altre parole, anche con una copertura di guasto molto elevata la qualità offerta ai clienti sarà bassa. Se però la resa è 0.8, basterà una copertura di guasto del 95% per raggiungere lo stesso *DL*. La relazione appena introdotta può essere rappresentata dal seguente grafico, dove le curve sono contraddistinte dal parametro *Y*.



Molti programmi per la generazione automatica dei vettori di test usano un simulatore di guasto per valutare un insieme di test proposto *T* e modificarlo (aggiungendo nuovi vettori e/o scartando vettori che non hanno migliorato la copertura) finché non si raggiunge una copertura di guasto soddisfacente: di fatto, la simulazione di guasto diventa quindi parte integrante della generazione dei test. Il procedimento può essere schematizzato come segue:



In alternativa, altri programmi generatori di test *generano un vettore di test mirandolo a un guasto obiettivo*: si verifica poi se questo test rileva anche altri guasti, li si scarta dall'insieme di guasti per cui si continua la ricerca di test (*fault dropping*) e si sceglie un altro guasto obiettivo fra quelli rimasti:



La simulazione di guasto può essere utilizzata anche per costruire un **dizionario di guasto,** che registri la risposta a T dei circuiti guasti N_f corrispondenti a tutti i vari guasti simulati *f*: di fatto, non si memorizza la risposta completa R_f, ma solo una opportuna funzione $S(R_f)$ detta *firma* o "signature" di *f*). Il dizionario di guasto può essere usato in particolare per la *localizzazione* dei guasti.

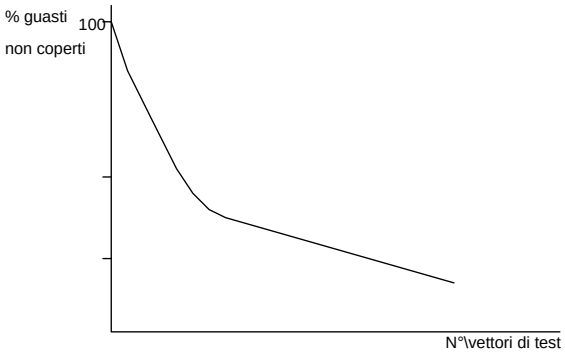
I simulatori di guasto sono correlati al *livello di astrazione* cui si pongono: tipicamente, i simulatori commerciali sono a livello logico o anche di transistore, mentre simulatori funzionali o comportamentali vengono eventualmente realizzati "ad hoc" per particolari dispositivi (o sistemi): le soluzioni più moderne sono di tipo *gerarchico*. I simulatori commerciali spesso forniscono anche indicazioni sui malfunzionamenti transitori (alee e corse) e indicano i *guasti più difficili da rilevare*.

Si vedranno ora in dettaglio alcune tecniche fondamentali di simulazione a livello di porta logica.

Che cosa significa, concettualmente, simulazione di guasto?

- Si dispone di una **simulazione logica** a livello di porta logica, a cui si fornisce la descrizione del circuito sano.
- si valuta, per un dato vettore di test, il comportamento del circuito sano.
- per ogni possibile guasto compreso nel modello dato (o per ogni classe di equivalenza) si valuta il comportamento del circuito in risposta allo stesso vettore di test;
- tutti i guasti tali che le uscite prodotte siano diverse da quelle prodotte dal circuito sano sono **rilevati** dal vettore di test in esame.

La ricerca dei vettori di test per un circuito dato, con il supporto della simulazione di guasto, è stata definita "sparare a un obiettivo che rimpicciolisce": all'inizio, quando nessun guasto è stato ancora coperto, anche un vettore scelto a caso coprirà uno o più guasti - man mano che la percentuale di guasti da rilevare diminuisce, però, una scelta casuale diventa sempre meno produttiva e anche l'identificazione algoritmica dei vettori necessari per completare la copertura diventa più difficile. Nella prima fase, anche una generazione *casuale* dei vettori di test può quindi essere produttiva (e risulta poco costosa): il numero di guasti ancora da rilevare diminuisce in modo *esponenziale* all'aggiunta di nuovi vettori. Questo vale fino a raggiungere una copertura che può andare dal 65 al 85%. In una seconda fase diventa necessaria una strategia mirata per la generazione dei vettori, e la curva che rappresenta la relazione fra numero di vettori generati e percentuale di guasti non coperti residui assume un andamento lineare (le cifre derivano da valutazioni empiriche realizzate ai Bell Labs.).



La simulazione ha senso se

1. il suo uso consente di ridurre il costo di *generazione* dei test (grazie al fault dropping)
2. si riduce il costo di applicazione del test (si realizzano solo tanti vettori quanti sono necessari, non tutti i vettori per i diversi guasti).

Occorre quindi che la complessità della simulazione sia polinomiale e non esponenziale. Si è valutato che il costo della simulazione per una rete con G porte cresca (a seconda del metodo) con G^2 o G^3; anche se il costo non è esponenziale, è comunque elevato per reti di rilevante complessità. Si considerano ora le tecniche di simulazione dei guasti più importanti; ovviamente, la *simulazione logica* ne è comunque la componente di base.

La simulazione di guasto più banale consiste in una **simulazione seriale:** si trasforma il modello del circuito sano N così da modellare il circuito N_f che contiene il guasto f, quindi si simula N_f e si ripete la procedura per tutti i guasti che interessano. I guasti vengono dunque simulati uno per volta; in realtà non occorre uno speciale simulatore di guasti - basta un normale simulatore logico - e si può simulare qualsiasi guasto purché ne sia noto il modello.

Questo permette di estendere la simulazione a un qualsiasi livello di astrazione. Lo svantaggio è il costo in termini di tempo di elaborazione.

Tutti gli altri schemi di simulazione di guasto qui considerati differiscono da questa soluzione perché:

- determinano il comportamento di *N* in presenza di guasti senza modificarne esplicitamente il modello (si "iniettano" I guasti, invece di modellare il circuito guasto);
- simulano contemporaneamente un insieme di guasti a ogni passata, rendendo quindi molto più veloce l'analisi per un vettore o un insieme di vettori di test.

I principali metodi di simulazione di guasto si distinguono in: **paralleli**, **deduttivi**, **concorrenti**. Si possono anche distinguere i simulatori di guasto - come i normali simulatori logici - in simulatori ***table driven*** **e** simulatori ***compiler driven***. I simulatori deduttivi e concorrenti appartengono alla classe ***event directed*** (*selective trace*) e sono del tipo table driven; quasi tutti i simulatori paralleli sono compiler driven.

Simulazione parallela

È stata la prima tecnica adottata per la simulazione di guasto (il primo simulatore di guasto commerciale, **TEGAS**, che risale al 1970 e fu molto noto e usato, era di tipo parallelo). Viene effettuata a livello di porta logica; si sfruttano le istruzioni logiche del calcolatore su cui si esegue la simulazione, che sono "bit-oriented" (operano sui singoli bit delle parole separatamente): es.: *[1010]AND[0111]=[0010]*; si opera quindi *in parallelo* (simultaneamente) su tutti i bit della parola (si è accennato a questo tipo di soluzione anche per I normali simulatori logici).

Date parole di memoria di *n* bit, si elaborano in parallelo *n* diversi problemi, definiti come diverse istanze di guasto sullo stesso circuito; se si vogliono simulare *m* guasti, per un dato vettore d'ingresso si dovranno compiere $\frac{m}{n}$ diverse passate di simulazione. Degli *n* bit della parola, uno corrisponde al circuito sano, *n-1* corrispondono ad altrettanti guasti distinti; se a ogni linea di segnale *S* si associa una parola di valutazione che fornisce il valore del segnale su *S* in corrispondenza del vettore di test considerato, saremo quindi n grado di verificare se (e quali) guasti iniettati a monte di *S* si propagano fino ad *S*.

Per simulare il guasto, si inietta nella computazione l'effetto logico del guasto (*iniezione del guasto*). A questo scopo, a ogni linea di segnale *S* si associano due maschere, *mask(S)* e *fvalue(S)*: se nella passata di simulazione sono inclusi anche guasti sulla linea *S*, a questi corrispondono opportuni valori dei bit delle due maschere associati ai guasti stessi. Più precisamente, si pone $mask(S)_i=1$ se sulla linea *S* esiste un guasto quando si simula il bit *i*-esimo, *=0* altrimenti; è poi $fvalue(S)_i=1$ se il guasto è uno *s-a-1*, $fvalue(S)_i=0$ se è uno *s-a-0*.

L'iniezione di guasti su *S* può essere vista come una particolare operazione che trasforma il valore di *S* (valutato simulando la porta logica che genera *S*) in un valore *S'* dovuto ai guasti, che viene ottenuto come $S' = S.\overline{mask(S)} + mask(S).\,fvalue(S)$

Le maschere da usare per l'iniezione dei guasti vengono generate in una fase di "preelaborazione" del circuito; a parte il tempo necessario per creare le maschere, gli *n-1* guasti vengono poi simulati nel tempo che sarebbe necessario per simularne uno.

Si voglia ad esempio simulare una porta NAND con due ingressi A e B e uscita C; supponiamo che sia *n=5*. L'associazione fra bit della parola e guasti sia la seguente:

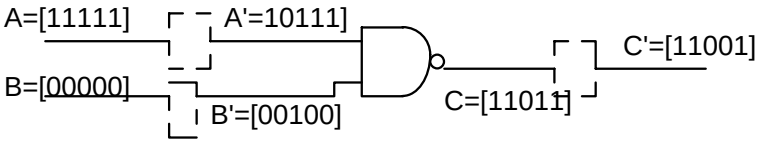
Posizione del bit	guasto
1	circuito sano
2	A s-a-0
3	B s-a-1
4	C s-a-0
5	C s-a-1

Si costruiscono le maschere per i quattro guasti considerati e per le varie linee:

Linea	Mask	Fvalue
A	[01000]	[00000]
B	[00100]	[00100]
C	[00011]	[00001]

(Ad esempio, è *mask(B)=[00100]* perché la posizione del guasto associato alla linea *B* è quella del terzo bit, e *fvalue(B)=[00100]* perché il guasto che si considera è uno *s-a-1*. Poiché sulla linea *C* si considerano due guasti, *C-s-a-0* e *C-s-a-0*, e ad essi si associano i bit di posizione 4 e 5, è *mask(C)=[00011]* e *fvalue(C)=[00001]*).

Una volta generate le maschere, si procede alla simulazione parallela per il vettore di test assegnato. Sia il vettore *AB=10*. Si inizializzano le linee di ingresso ai valori nominali: *A=[11111]* e *B=[00000]*. Si procede con la simulazione dagli ingressi verso le uscite, per livelli:



$$A' = [11111].\overline{[01000]} + [01000].[00000] = [10111]$$

$$B' = [00000].\overline{[00100]} + [00100].[00100] = [00100]$$

quindi si simula la porta NAND:

$$C = \overline{A'.B'} = [11011]$$

si applicano poi le maschere relative alla linea *C*:

$$C' = [11011].\overline{[00011]} + [00011].[00001] = [11001]$$

Si vede così che all'uscita del NAND i bit di posizione 3 e 4 sono diversi dal bit 1; il vettore *10* permette quindi di rilevare i due guasti *B s-a-1* e *C s-a-0*.

Simulazione deduttiva

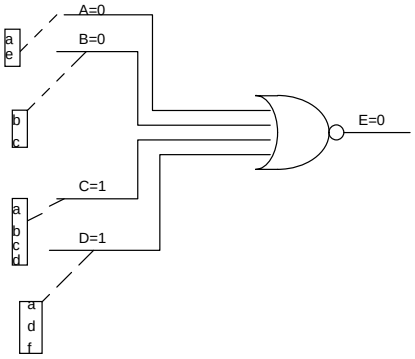
Introdotta nei primi anni '70, è stata utilizzata come base per il simulatore commerciale LAMP (Bell Labs). Il principio fondamentale consiste nel simulare esplicitamente solo il comportamento logico del circuito sano; al tempo stesso si deducono dallo stato "sano" *tutti* i guasti che potrebbero essere rilevati su qualsiasi linea di segnale o uscita del circuito dal vettore di test applicato.

Anche se il tempo richiesto dalla simulazione per un dato vettore è più lungo di quello necessario per una passata di simulazione parallela, comunque è in genere molto più ridotto di quello richiesto dalle $\frac{m}{n}$ passate di simulazione parallela che sarebbero necessarie per simulare

tutti i guasti. Di fatto, è stato dimostrato sperimentalmente che la simulazione deduttiva è più veloce di quella parallela per circuiti di grandi dimensioni.

Nella simulazione deduttiva, ad ogni linea di segnale A si associa una lista di guasti L_A che contiene il nome di tutti i guasti che producono un errore su A quando al circuito è applicato l'attuale vettore d'ingresso (o più in generale, prendendo in considerazione anche reti sequenziali, quando il circuito è nell'attuale stato logico). In altre parole: ogni guasto nella lista, se fosse il solo presente nel circuito, complementerebbe il valore di A. Le liste di guasti associate alle uscite primarie, al termine della simulazione, conterranno tutti i guasti rilevati dal vettore di test applicato. Le liste di guasti vengono propagate dagli ingressi primari verso le uscite primarie del circuito nel corso della simulazione. Inizialmente, cioè in corrispondenza dell'applicazione del primo vettore di test, le liste di guasti vengono *generate* per ogni linea; successivamente, quando si simulano nuovi vettori, ogni vettore introduce rispetto al precedente degli *eventi logici* (cambiamenti di valore sulle linee di segnale) e inoltre può provocare la modifica di alcune linee di guasto - può creare cioè degli *eventi di lista*. Nelle simulazioni successive alla prima, applicando un nuovo vettore di test lo stato di uscita di un elemento logico viene calcolato solo quando si verifica un *evento* (logico o di lista) su uno dei suoi ingressi. Il simulatore deve ricalcolare la lista di guasti associata all'uscita di un elemento sia in conseguenza di un evento logico, sia in conseguenza di un evento di lista a un ingresso dell'elemento stesso.

Si consideri ad esempio una porta NOR con quattro ingressi *A, B. C, D* e un'uscita *E*, facente parte di un circuito più complesso. Si supponga che per il vettore di test applicato i valori corretti degli ingressi siano 0011, e che il vettore stesso propaghi agli ingressi del NOR le liste di guasto indicate in figura (ai guasti si sono assegnati nomi simbolici).



Si voglia ora calcolare la lista di guasto associata ad *E*; per questo, si può notare che:

1. *a* è presente nelle liste di *A*, *C* e *D* - cioè complementa i valori di tutte e tre le linee; gli ingressi si portano quindi a 1000, e l'uscita resta a 0 (valore dell'uscita nel circuito sano). L'errore *a non* viene propagato alla lista di guasti associata a *E*;
2. *d* appare nelle liste di *C* e *D*; gli ingressi si portano a *0000*, l'uscita passa a 1: *d* viene inserito nella lista di guasti dell'uscita *E*;
3. per propagarsi all'uscita *E* del NOR un guasto deve non comparire nella lista di alcun ingresso il cui valore nominale è 0, e comparire nella lista di tutti gli ingressi il cui valore nominale è 1. In particolare, quindi, nessun guasto stuck-at su uno degli ingressi - per questo vettore di test - potrà propagarsi ad *E*.

Si ha

$$L_E = \overline{(L_A \cup L_B)} \cap L_C \cap L_D \cup E_{s-a-1}$$

Le liste di guasto che si propagano si calcolano utilizzando l'algebra degli insiemi; l'insieme di operazioni varia con il tipo di porta logica e con l'inversione di segnale che si deve propagare. Si consideri una porta AND a due ingressi: $Z=A.B$. Sia $A=B=1$: in assenza di guasto, $Z=1$. Qualsiasi guasto *s-a-0* su A o su B porta Z a 0: la lista di guasti associata a Z è: $L_z = L_A \cup L_B \cup \{Z_{s-a-0}\}$. Sia ora $A=0$, $B=1$: è $Z=0$. Un guasto che porti A a 1 senza modificare B cambia il valore di Z, mentre nessuno dei guasti che modificano B influenza Z (e un guasto che cambiasse sia A che B non cambierebbe Z). Quindi:

$$L_z = \left(L_A \cap \overline{L_B}\right) \cup \{Z_{s-a-1}\} = \left(L_A - L_B\right) \cup \{Z_{s-a-1}\}$$

dove il complemento di una lista indica l'insieme di guasti che *non* si trovano nella lista di origine. Ricordando i valori *controllanti* e *di inversione* per le principali porte logiche:

	c	i
AND	0	0
OR	1	0
NAND	0	1
NOR	1	1

si generalizza quanto detto come segue:

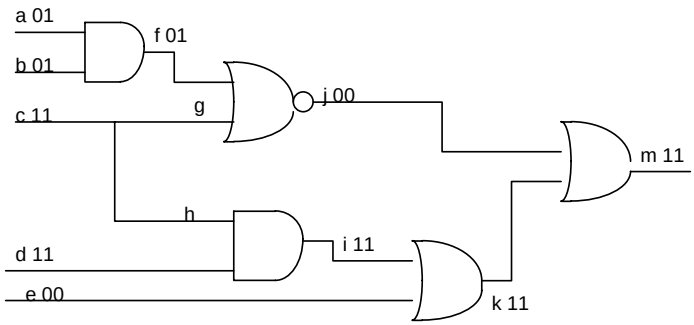
Per una porta Z con insieme di ingressi I, avente valore controllante c e inversione i, se C è l'insieme di ingressi con valore c la lista di guasti relativa a Z si calcola in base a:

$$\textit{if} \quad C = \varnothing \textit{ then } L_z = \left\{\bigcup_{j \in I} L_j\right\} \bigcup \{Z_{s-a-(c \oplus i)}\}$$

$$\textit{else } L_z = \left\{\bigcap_{j \in C} L_j\right\} - \left\{\bigcup_{j \in I} - CL_j\right\} \bigcup \left\{Z_{s-a-(\bar{c} \oplus i)}\right\}$$

cioè: se nessun ingresso ha valore c, qualsiasi effetto di un guasto si propaga all'uscita; se alcuni ingressi hanno valore c, solo un effetto di guasto che *tocca tutti gli ingressi con valore c senza toccare nessuno degli ingressi con valore opposto* si propaga alle uscite. In ambedue i casi, il guasto locale viene aggiunto alla lista dell'uscita.

Si analizzi ora un circuito più complesso di una semplice porta logica:



Si opera innanzitutto un *dominance fault collapsing*, al termine del quale restano le classi di guasto:

$$\{a_0, a_1, b_1, c_0, c_1, d_1, e_0, g_0, h_0, h_1\}$$

(di fatto, dato che i guasti dei checkpoint sono non ridondanti, si effettua il test appunto su tali guasti, come indicato a suo tempo).

Sia 00110 il primo vettore di test. Il calcolo delle liste procede come segue (in ogni lista si inseriscono solo - se è il caso - guasti appartenenti all'insieme rispetto a cui si effettua il test: le

liste risultano quindi ridotte rispetto a quelle che verrebbero calcolate con l'algoritmo prima indicato):

$L_a = \{a_1\}, \quad L_b = \{b_1\}, \quad L_c = \{c_0\}, \quad L_d = \varnothing, L_e = \varnothing$

$L_f = L_a \cap L_b = \varnothing, L_g = L_c \cup \{g_0\} = \{c_0, g_0\}, L_h = L_c \cup \{h_0\} = \{c_0 h_0\}$

$L_j = L_g - L_f = \{c_0, g_0\}, L_i = L_d \cup L_h = \{c_0, h_0\}$

$L_k = L_i - L_e = \{c_0, h_0\}$

$L_m = L_k - L_j = \{h_0\}$

La lista associata all'uscita m indica che il guasto h_0 viene rilevato: lo si cancella dall'insieme di guasti da simulare togliendolo anche da tutte le liste in cui compare, cioè L_k, L_i, L_h, L_m.

Si passa ora a simulare il comportamento del circuito in risposta al vettore di test 11110; c'è un evento logico sulle linee *a* e *b*. Si ottiene:

$L_a = \{a_0\}, L_b = \varnothing$; c'è un evento logico su *f* (diventa *f=1*) quindi si valuta $L_f = \{a_0\}$ La valutazione del NOR non genera eventi logici, ma ora è $L_j = L_f \cap L_g = \varnothing$: si vede cioè come sia possibile avere un evento di lista anche senza un evento logico corrispondente. Propagando l'evento di lista fino alla linea *m*, si ottiene $L_m = L_k - L_j = \{c_0\}$: il guasto c_0 viene quindi rilevato dal nuovo vettore di test. Vale la pena di sottolineare che, quando si calcola una nuova lista, per determinare se si è verificato un evento di lista occorre confrontarla con la lista omonima precedente: solo dopo il confronto, la precedente può essere distrutta.

La propagazione è notevolmente più complessa nel caso dei dispositivi di memoria; occorre infatti tenere in considerazione anche lo stato dei bistabili.

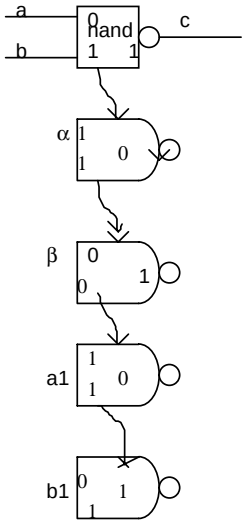
La complessità delle operazioni sulle liste dipende dalle strutture dati che si utilizzano; le varie alternative proposte nella letteratura mettono in evidenza un bilancio fra velocità di operazione e occupazione di memoria. In generale, comunque, non è possibile predire lo spazio di memoria richiesto, dato che si opera su liste dinamiche le cui dimensioni non sono valutabili a priori (in particolare, in presenza di anelli di reazione). La complessità della simulazione aumenta poi in modo molto rilevante se si deve ricorrere alla simulazione a 3 valori.

Simulazione concorrente.

Anche questa tecnica utilizza una sola passata per ogni vettore di test: si basa sull'osservazione che di solito (particolarmente per circuiti grandi) il comportamento del circuito con un guasto differisce di poco da quello del circuito sano, e che di conseguenza, quando si simula il circuito sano *N* per ogni circuito guasto N_f è sufficiente simulare solo quegli elementi in N_f che sono diversi dai corrispondenti elementi in *N*. Queste differenze vengono mantenute associando a ogni elemento *x* in *N* una **lista di guasti concorrenti,** CL_x. Per chiarire la tecnica di simulazione concorrente, si immagini inizialmente di costruire per ogni guasto *f* la corrispondente replica N_f del circuito, e di indicare con x_f la replica di *x* nel circuito N_f. Si indichi con V_x (e, rispettivamente, V_{xf}) l'insieme dei valori di ingressi, uscite (ed eventualmente stati interni) di *x* (x_f); durante la simulazione, CL_x rappresenta l'insieme di tutti gli elementi x_f, corrispondenti a diversi guasti, che differiscono da *x* nell'istante simulato attuale. Gli elementi x_f e *x* possono differire per due motivi: può essere $V_x \neq V_{xf}$ come

conseguenza della propagazione di un effetto di guasto provocato da *f* fino a una linea d'ingresso o uscita o a una variabile di stato di x_f, oppure *f* può essere un *guasto locale di* x_f - cioè un guasto inserito su una linea di ingresso o di uscita o in una variabile di stato dell'elemento x_f. Un guasto locale provoca una differenza fra x_f e x anche nel caso in cui fosse $V_x = V_{xf}$, cosa che succede se la sequenza d'ingresso applicata non attiva *f*.

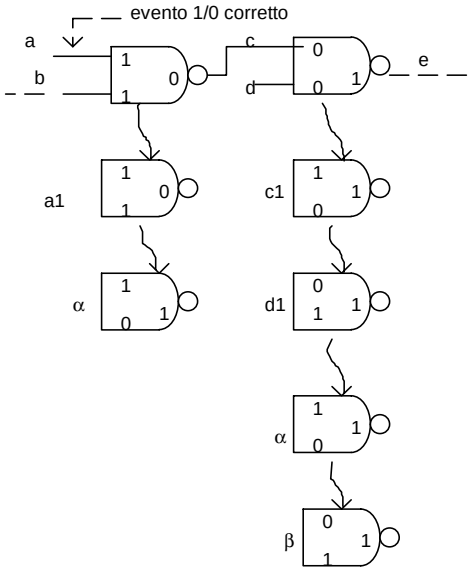
Un elemento nella lista CL_x ha la forma (f, V_{xf}), cioè consta dell'indicazione del guasto e dell'insieme dei valori associati a x in corrispondenza. In figura si dà una rappresentazione grafica di una lista di guasto concorrente, per una porta NAND con linee d'ingresso *a,b* e linea d'uscita *c*; il NAND più in alto è inserito nel circuito sano, e l'insieme di valori associato è *abc=011*, mentre le copie "appese" sono quelle che corrispondono ai diversi guasti α, β (propagati fino a qui) e a *a* s-a-1, *b* s-a-1 (guasti locali). (Anche qui si suppone di avere effettuato un dominance fault collapsing, per cui non si considera il guasto *c* s-a-0). Si noti come si sia preso in considerazione anche il guasto *b* s-a-1, che *non* è attivato dall'attuale vettore; la lista CL_x è notevolmente più complessa e articolata di quella che si realizzerebbe nel caso di simulazione deduttiva, e che comprenderebbe solo α e a_1. Sulla destra è riportata la forma tabellare della lista di guasto.



f	a	b	c
α	1	1	0
β	0	0	1
a1	1	1	0
b1	0	1	1

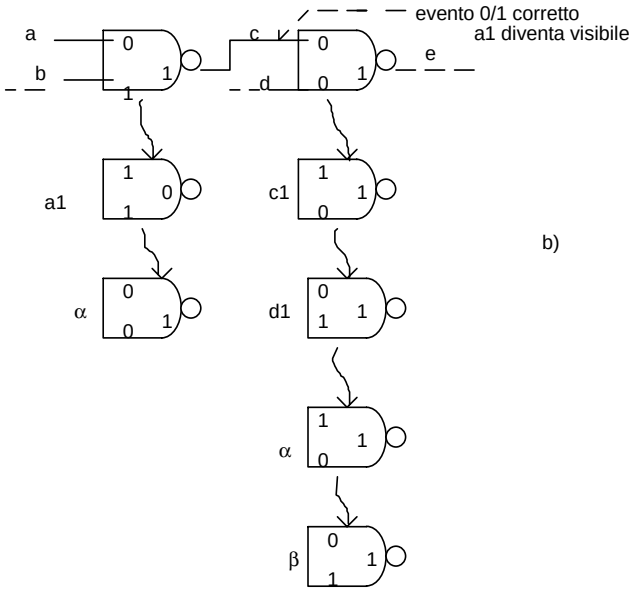
Si dice che un guasto *f* è *visibile* su una linea *i* quando i valori di *i* in *N* e in N_f sono diversi; solo i guasti visibili in CL_x per la porta x apparirebbero nella lista deduttiva. Anche le liste concorrenti di guasto sono strutture *dinamiche*, che vengono aggiornate nel corso della simulazione. All'inizio, della simulazione, si opera l'inserimento dei guasti: per ogni lista CL_x la lista contiene solo i guasti locali; che vi restano solo fino a quando vengono fatti "cadere" dalla lista dei guasti ancora da coprire; nel corso della simulazione, i nuovi elementi in CL_x rappresentano elementi x_f i cui valori diventano diversi da quello di x (*divergono* da quello di x), mentre gli elementi che hanno valore uguale a x vengono rimossi.

Si considerino ad esempio le liste di guasto per un segmento di circuito a partire da uno stato "stabile". Inizialmente (fig. a) agli ingressi primari *a, b* e *d* sono applicati i valori 110; come sempre, si è operato un *dominance fault collapsing*. Siano α, β guasti propagati e *a*1, *c*1, *d*1 guasti locali. Il guasto *a*1 non si propaga dal primo al secondo NAND.



a)

A questo punto, si applica un nuovo vettore di test: l'ingresso primario *a* cambia da 1 a 0 generando un evento (corretto) 1/0. L'evento si verifica anche (implicitamente) in tutti i circuiti guasti per cui non compaiono elementi in *CLc* (questa lista sarà esaminata a parte). Nel circuito sano, l'evento su *a* fa passare *c* da 0 a 1: l'evento su *a* non si propaga nel circuito *a1*, quindi questa porta non viene valutata. La valutazione della porta α non produce alcun evento. (fig. b)



b)

L'evento corretto su *a* provoca un evento corretto 0/1 sull'ingresso *c* del secondo NAND; si aggiorna quindi la lista concorrente del secondo NAND. Il guasto *a*1 ora diventa visibile sulla linea *c*; la lista di guasto associata al secondo NAND viene di conseguenza modificata. Nel circuito sano, la variazione di *c* si propaga alla porta *e*. Si propaga anche un "evento di lista" per indicare che *a1* è diventato visibile sulla linea *c*.

La valutazione di *e* nel circuito sano non produce eventi. A CL_e si aggiunge un elemento relativo a *a1*. Si analizzino ora gli altri elementi in Cl_e (fig. c):

c non cambia nel circuito *c1*: l'elemento per *c1* resta nella lista perché si riferisce a un guasto locale della porta *e*;

- la porta *e* nel circuito *d1* viene valutata e genera un evento 1/0: lo stesso accade per il circuito β;
- *c* non cambia nel circuito α: si cancella dalla lista l'elemento relativo ad α.

Il meccanismo di simulazione concorrente valuta i singoli elementi sia nel circuito sano che in quelli guasti; un elemento in una lista di guasto che indica la replica di una porta del circuito sano è valutato come una normale porta con ingressi diversi.

La simulazione concorrente si basa sulla valutazione dei singoli elementi, e può essere compiuta anche ad altri livelli di astrazione - in particolare, a livello funzionale. Lo svantaggio principale è la quantità di memoria occupata, maggiore di quella richiesta dalla simulazione deduttiva. In ambedue le tecniche, a priori non è possibile stabilire la quantità di memoria richiesta prima di una passata; occorre quindi un sistema capace di allocazione dinamica della memoria.

TECNICHE DI TESTING

Si esamineranno ora alcune tecniche fondamentali per la generazione dei vettori di test.

Gli approcci al collaudo possono distinguersi in base alle tecniche usate per generare e per elaborare i risultati del test: una prima suddivisione distingue:

- Test ***On line***: eseguito *durante il funzionamento* del sistema (a "run time"), usando i normali dati d'ingresso come vettori di test; l'errore provocato da un guasto può essere rilevato da appositi circuiti di controllo (monitoring circuits) associati al sistema. Viene detto anche *test concorrente*, in quanto effettuato simultaneamente alle normali operazioni del sistema.
- Test ***Off line***: eseguito quando il sistema sotto test *non* sta eseguendo le sue funzioni nominali; può essere compiuto sia al termine della produzione, sia in esercizio - in quest'ultimo caso, però, il funzionamento normale viene sospeso per consentire in alternativa le attività di collaudo.

.Un'altra distinzione possibile vede:

- Test ***esterno***: la strumentazione che opera il test è esterna al sistema sottoposto al test - si tratta di macchine apposite (*Automatic Test Equipment*, ATE) che applicano i vettori di test al sistema sotto test e ne rilevano le risposte, analizzandole direttamente o registrandole per un'analisi successiva;
- ***Built-In Testing***: la strumentazione è inserita *nel sistema sotto test* stesso. Questa tecnica viene tipicamente usata per il test *on line*, ma in un numero crescente di casi viene adottata anche (almeno parzialmente) per il test off-line di sistemi o dispositivi complessi.

Il test *off line* è quindi realizzabile sia come test esterno, sia come built-in-test; il test *on line* richiede inevitabilmente l'introduzione di una qualche forma di *ridondanza*. (Si parla di ridondanza *strutturale* quando la soluzione prevede essenzialmente l'introduzione di strutture circuitali ridondanti rispetto a quella nominale, mantenendo le prestazioni temporali il più prossime possibile a quelle nominali; di ridondanza *temporale* se la tecnica sfrutta l'esecuzione ripetuta di determinate operazioni da parte di unità diverse e il successivo confronto dei risultati, riducendo in modo rilevante la frequenza di generazione dei risultati ma con l'introduzione di modeste ridondanze strutturali; di ridondanza *di informazione* quando si ricorre a forme di codifica per il rilevamento degli errori).

In questo corso si tratterà essenzialmente il problema del test *off-line*; si farà riferimento dapprima a quello *esterno*, e successivamente si discuteranno le soluzioni più diffuse per quello *built-in*.

Test esterno off-line

I parametri rispetto a cui si valuta un insieme di vettori di test sono:

a) la **copertura** offerta dal test
b) il costo di **applicazione** del test, normalmente misurato come numero di vettori di test che devono essere applicati per garantire una certa copertura (quanto maggiore è questo numero, tanto maggiore è il tempo d'uso della strumentazione di test e il tempo di "fermata" del dispositivo sotto test: ambedue questi fattori contribuiscono al costo);

c) il costo di **generazione** del test, normalmente valutato in rapporto alla complessità computazionale dell'algoritmo usato per generare un insieme di vettori di test che garantiscano una data copertura, e quindi al tempo di calcolo necessario per la generazione.

La generazione dei vettori di test viene effettuata di solito usando appositi programmi (*ATPG*, *A*utomatic *T*est *P*attern *G*enerator) su calcolatori di tipo generale; successivamente, tali vettori vengono applicati da apparecchiature di test (*ATE*, *A*utomatic *T*est *E*quipment) che permettono anche la *valutazione* del test, cioè - in definitiva - il rilevamento dei guasti. Alcuni sistemi ATE si basano sul concetto di *gold unit*: nella macchina di test vengono inseriti un esemplare del sistema sotto test "notoriamente buono" (*known good*) e quello sottoposto a test, e ambedue ricevono simultaneamente in ingresso i vettori di test: i risultati prodotti dai due esemplari vengono immediatamente confrontati e una discrepanza porta a dichiarare guasto il sistema sotto test. Questa soluzione permette di agire velocemente e con costi relativamente modesti (spesso opera a livello *funzionale* o addirittura *comportamentale* - è molto usata per sistemi a microprocessore); è però fortemente specializzata su una particolare struttura di sistema sotto test (o su una classe di sistemi). In alternative, macchine di test più generali (e molto più costose) prevedono l'applicazione dei vettori di test (in sequenza nota) al solo sistema sotto test, la memorizzazione dei risultati e la successiva verifica per confronto con i risultati forniti dalla simulazione logica.

I tre parametri di costo sono (almeno parzialmente) contrastanti: es.: per una rete combinatoria, un test esaustivo (costituito da tutte le possibili combinazioni d'ingresso) garantisce la massima copertura e minimizza il costo di generazione, ma dà anche il massimo costo di applicazione. Si noti che il costo di generazione viene affrontato una sola volta, al momento della generazione dei vettori di test; il costo di applicazione si ripete ogni volta che si sottopone a test un circuito.

Si affronterà dapprima il problema del collaudo di circuiti combinatori relativamente a guasti logici; si considererà poi il test delle macchine sequenziali, e si accennerà al problema del test funzionale.

Il test dei circuiti combinatori

- molto più semplice del caso sequenziale - non occorre informazione sullo stato in cui si trova il circuito;
- indispensabile anche per il test delle reti sequenziali.

Una soluzione banale, come già accennato, consiste nel *test esaustivo*: dato un circuito con n ingressi, si applicano tutte le 2^n configurazioni d'ingresso e per ognuna di verifica se la configurazione di uscita è identica a quella prodotta dal circuito sano. Questa soluzione dà la *massima copertura possibile* (i guasti che non si rilevano sono solo quelli che non generano errore sulle uscite, cioè sono ridondanti); di fatto, però, è accettabile solo per circuiti con n piccolo: il numero dei vettori e dei risultati - e quindi il tempo di applicazione del test e il tempo di valutazione - aumentano in modo esponenziale col numero degli ingressi. Peraltro, si ottimizzano copertura e costo di generazione; la generazione richiede semplicemente un contatore binario modulo 2^n.

Può diventare accettabile quando il circuito è *partizionabile* in sottocircuiti più semplici e mutuamente indipendenti, sottoposti in parallelo al test; di fatto, si tratta allora di modificare il progetto per potere compiere queste operazioni. (Val la pena di ricordare che partizionare un circuito dato è un problema di grande complessità computazionale!)

Tecniche esaustive vengono in realtà adottate in soluzione di *built-in self testing*: i vettori di test (in questo caso, tutti i vettori possibili) vengono generati automaticamente da un opportuno sottosistema, e i risultati vengono valutati automaticamente da un altro sottosistema, inserito nel progetto e attivato (come il primo) quando il sistema viene portato in uno stato di collaudo.

Il test casuale (Random Testing)

La generazione casuale dei vettori di test è un'altra tecnica poco costosa per generare i vettori di test; come il test esaustivo, è legata solo al numero degli ingressi e non alla topologia del circuito. I vettori di test vengono generati fuori linea mediante un processo casuale; la loro applicazione, poi, è deterministica, per consentire la verifica dei risultati. Se si vuole usare una soluzione totalmente casuale, per ottenere un test di alta qualità occorre un numero molto elevato di vettori di test generati casualmente. Si può inserire una tecnica di generazione casuale nella catena normale di generazione del test, se non altro, per verificare il grado di copertura raggiunto; dato il gran numero di vettori che si devono usare, aumentano i costi di *simulazione* e di *applicazione* dei vettori di test.

La tecnica di generazione casuale può essere implementata in qualsiasi linguaggio di programmazione, senza bisogno di usare un particolare linguaggio di descrizione dei circuiti; se si effettua la simulazione di guasto per ogni vettore generato, il procedimento di simulazione e verifica scarta poi qualsiasi vettore non aumenti la copertura (riducendo, almeno in parte, il costo di applicazione). Si è dimostrato che per reti combinatorie il metodo casuale dà risultati dipendenti dal numero di livelli e dal fan-in delle porte. Nel caso di reti sequenziali, la generazione non può comunque essere puramente casuale (es.: i segnali di clock non possono essere casuali; gli ingressi di dati devono adeguarsi a particolari restrizioni, quali le relazioni con i fronti del clock, per evitare corse; un vettore che non aumenta la copertura non può essere scartato immediatamente perché potrebbe portare la rete in un nuovo stato non prima raggiunto, etc.). Peraltro, la generazione casuale può essere spostata a qualsiasi livello di astrazione: es. per compiere il test di un microprocessore si possono generare in modo casuale le istruzioni che si applicano al microprocessore e le configurazioni di dati su cui ogni istruzione deve operare.

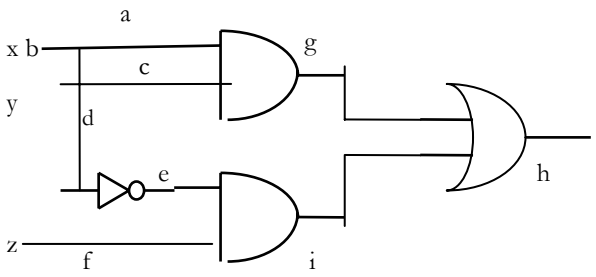
Il test dei circuiti combinatori: tecniche deterministiche

I generatori di test deterministici producono insiemi di vettori di test partendo dalla descrizione del circuito. Richiedono più tempo di elaborazione dei precedenti, ma producono insiemi di vettori di test più brevi e (rispetto a quelli casuali) di qualità più elevata. Possono essere *orientati al guasto* (si generano vettori di test per specifici guasti appartenenti all'universo dei guasti definito dal modello) o *indipendenti dal guasto* (operano senza mirare alla copertura di un guasto specifico).

Una soluzione esatta per generare l'insieme di test minimo a massima copertura: l'estensione del metodo di Quine-McCluskey

Si tratta di un metodo indipendente dal guasto: dato un circuito, si identificano le classi di guasti equivalenti e si valuta - per ogni possibile configurazione degli ingressi primari - quali di tali classi vengono rilevate applicando come vettore di test la configurazione stessa. Si definisce come insieme ottimo di vettori di test un insieme minimo di vettori di test che consenta di

rilevare tutti i guasti. Per trovare un tale insieme si ricorre a una *tabella di copertura* nella quale alle righe si associano le configurazioni degli ingressi e alle colonne le classi di guasto. L'elemento (*r,c*) viene marcato se il vettore di test *r* permette di rilevare un guasto appartenente alla classe *c* (cioè se, qualora fosse presente il guasto, l'uscita che si ottiene applicando *r* in ingresso sarebbe complementata rispetto al valore fornito dal circuito sano). Il metodo per determinare una copertura minima della tabella deriva direttamente da quello di Quine-McCluskey per la sintesi minima a due livelli delle reti combinatorie. Si consideri ad esempio il seguente semplice circuito:



Il circuito realizza la funzione $F = xy + \bar{x}z$; le classi di guasti equivalenti sono:

1) a/0, c/0,g/0; **2**) b/0; **3**) b/1; **4**) a/1; **5**) c/1; **6**) d/0,e/1; **7**) d/1,e/0,f/0,i/0; **8**) f/1; **9**) g/1,h/1,i/1; **10**) h/0

La tabella di copertura ha otto righe (le combinazioni d'ingresso) e dieci colonne (classi di guasto).

Xyz	1	2	3	4	5	6	7	8	9	10
000								x	x	
001			x				x			x
010				x					x	
011		x		x		x			x	
100			x		x			x	x	
101			x				x			x
110	x	x								x
111	x	x								x

Le colonne 5 e 6 contengono ognuna una sola marca, rispettivamente nelle righe 011 e 100; i due vettori 011 e 100 sono *essenziali* per garantire copertura del 100%, e vanno quindi scelti. Questi vettori coprono anche le classi 2, 4, 9, 3, 8, che non è necessario esaminare ulteriormente e possono essere eliminate (si mira a *rilevare* I guasti, non a localizzarli). Si passa a una tabella ridotta, in cui appaiono le sole colonne 1,7,10 non ancora coperte e le sei righe residue:

xyz	1	7	10
000			
001		x	x
010			
101		x	x
110	x		x
111	x		x

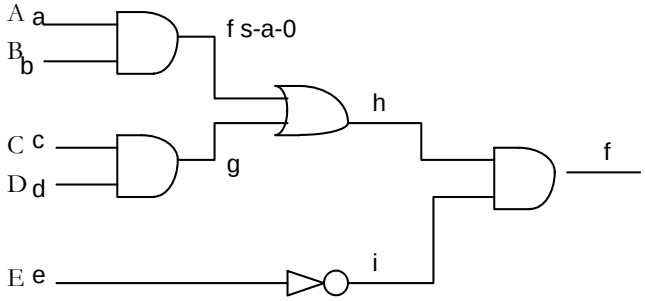
La colonna 10 è *dominante* rispetto alla 1 e alla 7: qualunque vettore copra 1 o 7, cioè, copre anche 10. La *colonna dominante* può essere eliminata, dato che la scelta di un vettore che copra 1

o 7 garantisce anche la copertura di 10 (ma non viceversa). Le scelte (001,110), oppure (001,111), o (101,110), o(101,111) per la copertura di 1 e 7 sono tutte di ugual costo e, unite ai due vettori essenziali, consentono la copertura al 100%. In questo circuito non esistono guasti ridondanti.

Occorre notare che, sebbene questo metodo porti all'ottimo, in genere, anche dopo l'applicazione delle relazioni di essenzialità e dominanza, resta un gran numero di possibili scelte da esplorare per garantire una soluzione ottima: in effetti, il problema della copertura è *NP completo* (non esiste cioè un algoritmo risolutivo di complessità polinomiale). È quindi evidente che la tecnica risulta inaccettabile per circuiti con un numero di ingressi e di linee anche ridotto.

Gli ATG orientati al guasto

In questi algoritmi, si seleziona un guasto *s-a-v* (rappresentativo di una classe): si tratta quindi di *attivarlo* (o *eccitarlo*) e quindi di *propagarlo alle uscite primarie* (PO). Eccitare un guasto significa imporre ingressi primari PI tali che, in assenza del guasto, sulla linea *l* cui è associato il guasto in questione sia presente il valore $\overline{v}$. Questa operazione si dice anche di *giustificazione della linea*. Eccitare il guasto e propagarne gli effetti è molto semplice nel caso di un circuito privo di fanout: ad esempio:



per eccitare il guasto *f s-a-0* occorre portare 1 sulla linea *f*: questo si ottiene con *A=B=1*. Per propagare il valore di *f* verso l'uscita, occorre che l'ingresso *g* dell'OR sia al valore non controllante, cioè 0, e che il valore dell'ingresso *i* all'AND finale sia al valore non controllante, e cioè 1. Si ricava un vettore di test *(ABCDE)=110x0* (in alternativa, anche *11x00*).

Per considerare la propagazione dei guasti si devono considerare i valori sia nel circuito sano *N* che in quello guasto N_f relativo al guasto *f* per cui si vuole generare un test. Per questo si introduce la definizione dei *valori logici compositi*, della forma v/v_f dove *v* e v_f sono i valori dello stesso segnale in *N* e in N_f. I valori logici compositi che rappresentano gli errori (1/0 e 0/1) si indicano con i simboli *D* e $\overline{D}$; gli altri due valori (0/0 e 1/1) si indicano come 0 e 1. Si ha cioè:

v/v_f	
0/0	0
1/1	1
1/0	D
0/1	$\overline{D}$

Un'operazione logica fra due valori compositi si fa operando separatamente sul valore corretto e su quello errato e poi componendo i risultati. Es.:

$\overline{D}+0=0/1+0/0=0+0/1+0=0/1=\overline{D}$

Si verifica che D si comporta secondo le regole dell'algebra di Boole, e cioè che $D+\overline{D}=1$, $D.\overline{D}=0$; $D+D=D.D=D$, $\overline{D}+\overline{D}=\overline{D}.\overline{D}=\overline{D}$. Si possono creare tabelle degli operatori booleani applicati a variabili che abbiano i cinque valore 0,1,D, $\overline{D}$, x. . È:

AND	0	1	D	$\overline{D}$	x
0	0	0	0	0	0
1	0	1	D	$\overline{D}$	x
D	0	D	D	0	x
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	x
x	0	x	x	x	x

OR	0	1	D	$\overline{D}$	x
0	0	1	D	$\overline{D}$	0
1	1	1	1	1	1
D	0	1	D	1	x
$\overline{D}$	$\overline{D}$	1	1	$\overline{D}$	x
x	x	1	x	x	x

Si consideri dapprima il caso di un circuito privo di fanout, la generazione di un test per un generico guasto *s-a-v* sulla linea *l* consiste nell'inizializzare i valori di tutte le linee del circuito (all'infuori di *l*) a x e compiere poi due passi fondamentali - realizzati dalle routines *Justify* e *Propagate* nel seguente schema di algoritmo - fino a determinare il (i) vettore(i) di test:

begin

inizializza tutti i valori a x

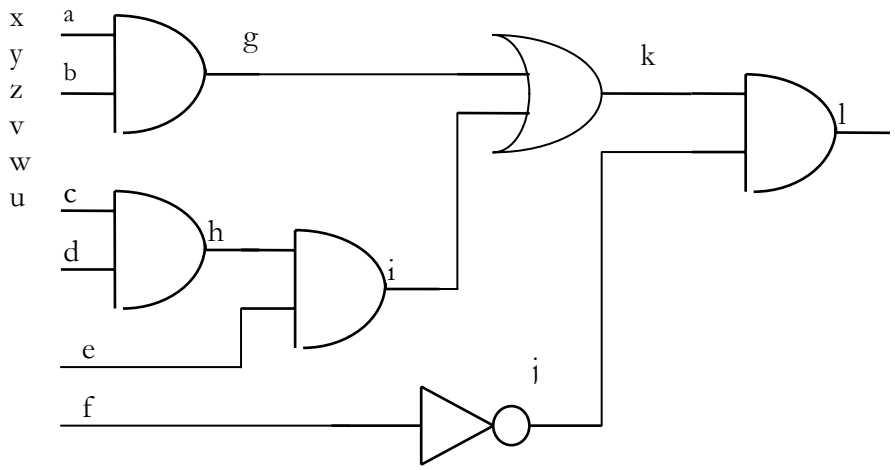
Justify(l, $\overline{v}$ *)*
if *v=0* **then** *Propagate (l,D)*

else *Propagate (l,* $\overline{D}$ *)*

end

La giustificazione di una linea è un processo ricorsivo: si giustifica il valore di uscita di una porta mediante i valori d'ingresso della porta stessa e si itera il procedimento fino a raggiungere gli ingressi primari. Ad esempio, per un NAND a k ingressi: si giustifica 0 in uscita solo se tutti gli ingressi sono a 1, mentre un 1 in uscita viene giustificato da 2^k-1 diverse configurazioni - la soluzione più semplice in questo caso consiste nell'assegnare *0* a un ingresso arbitrariamente scelto e x a tutti gli altri. Più in genere, si verifica se il valore sull'uscita della porta logica corrisponde a quello determinato da un valore controllante su un ingresso (in questo caso si fissa al valore controllante un ingresso arbitrariamente scelto, mentre gli altri sono lasciati a x e non richiedono ulteriore giustificazione) oppure da valori non controllanti su tutti gli ingressi (che saranno a loro volta giustificati). Per propagare l'errore fino alle uscite primarie, si deve *sensibilizzare un percorso* da *l* alle uscite primarie: nel caso di un circuito privo di fanout, tale percorso è unico, e ogni porta su di esso ha uno e un solo ingresso sensibilizzato al guasto, mentre tutti gli altri ingressi devono avere valore non controllante (la propagazione richiede a sua volta un insieme di giustificazioni). Di fatto, si implementa il concetto di **sensibilizzazione del percorso (Path Sensitizing)**, molto semplice se applicato a un *singolo* percorso di segnale in un circuito e costituito dalle due fasi di ***propagazione dell'errore (forward trace)*** (lungo il percorso dal guasto alle uscite, tutti gli ingressi alle porte all'infuori di quello proveniente dal guasto vengono posti a valori non controllanti - il percorso è *sensibilizzato*) e di ***backward trace*** o **giustificazione delle linee** (si procede all'indietro per garantire che i valori assunti sugli ingressi delle varie porte nel percorso sensibilizzato siano giustificati dai valori agli ingressi delle porte a monte; risalendo fino agli ingressi primari).

Si voglia rilevare il guasto *g s-a-0* nel seguente circuito:

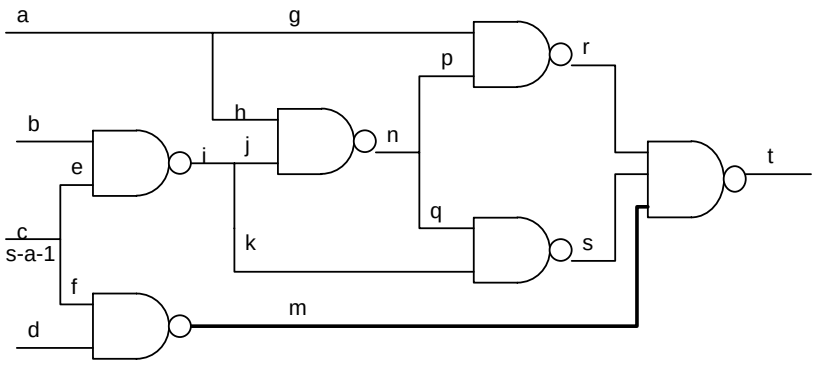


Per portare il valore 1 sulla linea *g*, si deve porre *xy=1*. La fase di *forward trace* crea il percorso di propagazione *k-l*; occorre che la linea *i* sia al valore non controllante per un OR (quindi 0) e la linea *j* al valore non controllante per un AND (quindi 1).

Occorre ora giustificare tali valori, procedendo alla fase di *backward trace*. *i=0* richiede valore controllante su almeno un ingresso - si scelga *e=0*: la linea *h* resta al valore *x*, e quindi non occorre procedere alla sua generazione (anche gli ingressi *z* e *v* restano a *x*). Per giustificare 1 su *j*, occorre che sia *f=0*, quindi *u=0*; l'insieme di vettori di test è 11*x*x00.

Il problema diventa nettamente più complesso quando sono presenti dei fanout. Anche in questo caso, la procedura consiste essenzialmente nell'attivare il guasto e propagarlo fino alle uscite, ma possono esistere più percorsi alternativi dal guasto a un'uscita primaria. Una volta scelto un percorso, la propagazione torna a risolversi in un insieme di problemi di giustificazione che però - se esistono fanout riconvergenti - non sono più indipendenti l'uno dall'altro come nel caso di un circuito privo di fanout. Inoltre, la presenza di fanout riconvergenti può portare a *conflitti*: può succedere che nella fase di giustificazione si tenti di imporre due valori opposti sulla stessa linea (o sullo stesso ingresso primario). In presenza di un conflitto, se si era operata una scelta e rimangono ancora alternative da esplorare, si ricorre a una strategia di backtracking che consenta un'esplorazione sistematica dello spazio delle soluzioni, superando la scelta non corretta. Altrimenti, il guasto in esame non può essere rilevato ricorrendo a una tecnica di sensibilizzazione di percorso singolo (ma, come si vedrà fra poco, questo non consente ancora di stabilire che il guasto sia ridondante).

Si consideri dapprima il seguente esempio: si voglia determinare un test per il guasto *c s-a-1*



Primo passo: eccitazione del guasto: sulla linea *c* si impone il valore 0. Per ottenerlo, qui basta porre l'ingresso primario C a 0.

Secondo passo: propagazione dell'errore verso un'uscita primaria. Si scelga dapprima il percorso *f-m-t*. Per sensibilizzarlo, occorre che *d* sia non controllante (D=1), e che *r* e *s* siano

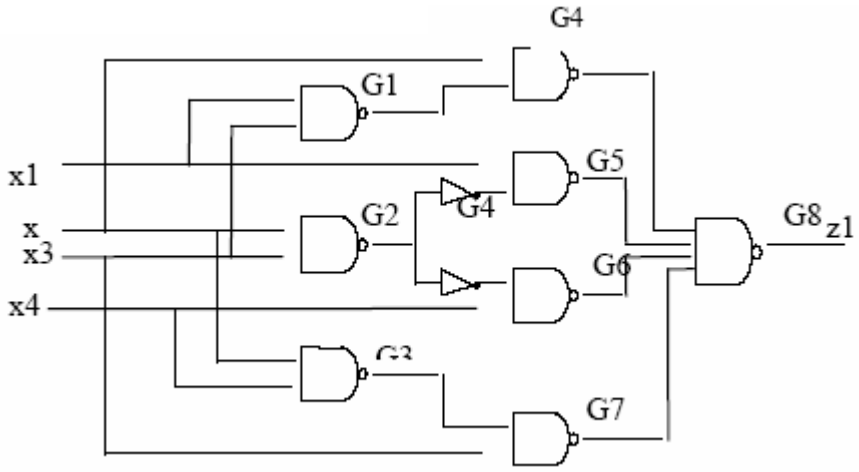
ambedue non controllanti (*r*=*s*=1). Questo garantisce che l'inversione di valore di *c* provochi un'inversione di valore di *t*. A questo punto (noti D=1 e C=0) si ha: *m=1, i=1, j=1, k=1, t=0*;

Terzo passo: giustificazione dei valori delle linee: si devono definire valori degli ingressi primari che giustifichino i valori imposti per i segnali interni; Perché sia *s=1* deve essere *q=0*, quindi è anche *p=0, n=0*. *p=0* dà la giustificazione per *r=1*. Perché sia *n=0* deve essere *h=1*, quindi anche *g=1* (consistente con *r=1*, essendo *p=0*) e *a=1*. La condizione *i=1* può essere soddisfatta ponendo *b=0* (quindi B=0).

Si noti che ponendo B=1 si sarebbe ancora generato un test valido, attivando questa volta *due* percorsi - oltre a *m-t* anche *i-k-s-t*

Ogni volta che si fa una scelta, si devono memorizzare le alternative scartate per l'eventualità di dover ripercorrere a ritroso il circuito (se l'alternativa scelta non conduce a buon fine) ed esplorare un'altra possibilità. Il metodo euristico utilizzato per compiere le scelte: determina le caratteristiche del particolare algoritmo di test in esame. Nei casi più sfortunati, occorre esaminare tutte le alternative possibili a ogni punto di scelta prima di determinare un vettore di test o eventualmente stabilire che il guasto scelto non è testabile. E' stato dimostrato che la generazione dei vettori di test, nel caso generale, è comunque un problema NP-completo.

Si consideri ora il seguente esempio:



si voglia rilevare un guasto *s-a-1* sull'uscita di G2.

Eccitazione del guasto: si deve imporre G2=0 ⇒ *x2=x3=1*.

Propagazione del guasto: Qualsiasi percorso singolo da G2 all'uscita passa o attraverso G5 o attraverso G6.

Si voglia dapprima sensibilizzare *il solo percorso* attraverso G5 e G8: si deve porre *x1=1* e imporre G4=G6=G7=1. G6=1 richiede *x4=0*; ma *x2=x3=1* e *x4=0* implicano G7=0; si ha cioè un'inconsistenza durante il passo di line justification. Un problema simmetrico sorge se si vuole sensibilizzare un singolo percorso attraverso G6 e G8.

Il guasto in esame però *non è* un guasto ridondante: il vettore d'ingresso 1111, che sensibilizza *simultaneamente* ambedue i percorsi da G2 all'uscita, permette di rilevare il guasto.

Il metodo usato per generare i vettori di test deve permettere di sensibilizzare simultaneamente tutti i possibili percorsi dal punto di guasto alle uscite del circuito. La sensibilizzazione multipla

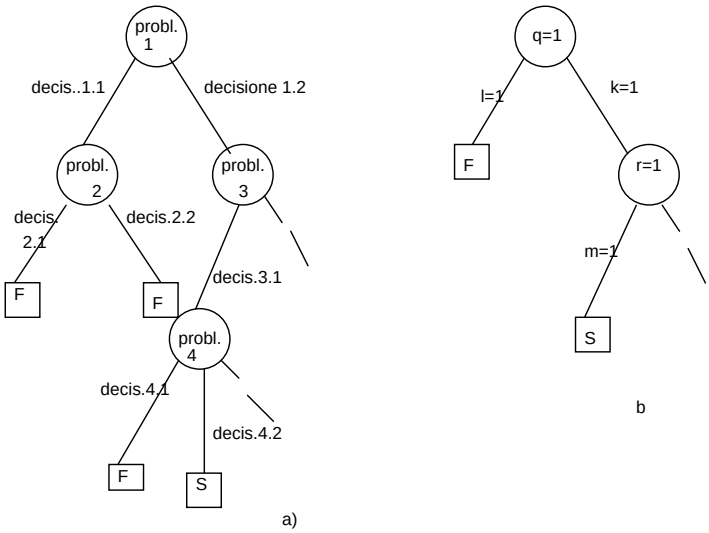
è stata introdotta da Roth con il *D-algorithm* (1966) ed è stata poi adottata dagli algoritmi successivi, basati su diverse euristiche e che fanno uso tutti della D-algebra.

Usando l'estensione dell'algebra booleana a variabili che assumono i valori 0, 1, *x*, *D*, $\overline{D}$, si introducono i concetti di *D-frontiera* e *J-frontiera*, dove:

- la *D-frontiera* è costituita (in un generico istante) da tutte le porte logiche la cui uscita è ancora indeterminata ma che hanno uno o più ingressi al valore *D* o $\overline{D}$. Nella *propagazione* dei guasti, si sceglie una porta sulla D-frontiera e si assegnano dei valori agli ingressi non specificati in modo da ottenere *D* o $\overline{D}$ sull'uscita (operazione detta anche *D-drive*). Se a un certo punto non è possibile propagare l'errore alle uscite primarie, occorre ricorrere a una fase di backtracking (se esistono ancora alternative non valutate) e ripetere la propagazione;
- la *J-frontiera* è costituita da tutte le porte che hanno uscita nota e determinata ma I cui ingressi devono ancora essere giustificati.

Sulla base di queste definizioni, gli algoritmi orientati al guasto procedono alla *implicazione*: si usano i primi valori assegnati per calcolare tutti I valori che possono essere implicati in modo univoco, si verifica la consistenza dei risultati, se necessario si assegnano nuovi valori (operando eventuali scelte), si ricalcolano D-frontiera e J-frontiera. Quando non risulta possibile propagare il guasto o garantire la consistenza nella fase di implicazione, si ritorna su una delle scelte operate in precedenza e per cui non si erano ancora esplorate tutte le alternative e si ripete la procedura: a questo scopo, si utilizza l'***albero delle decisioni*** (v. figura).

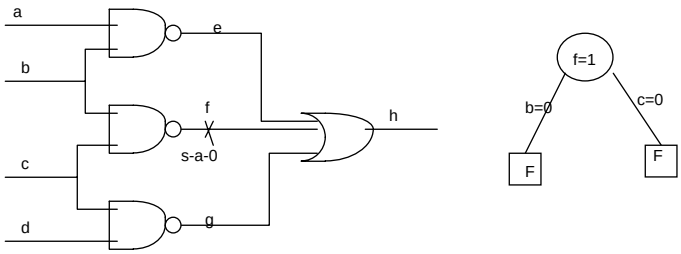
Un *nodo di decisione* (un cerchio nella figura) indica un problema che l'algoritmo tenta di risolvere: un ramo uscente dal nodo indica una decisione - la scelta di un'alternativa per risolvere il problema. Un nodo terminale marcato F (Fallimento) indica che si è trovata un'inconsistenza o che non si riesce a propagare ulteriormente l'errore verso un'uscita primaria. Un nodo terminale marcato S (Successo) indica che si è trovato un vettore di test. L'esecuzione dell'algoritmo può essere seguita con un attraversamento depth-first dell'albero delle decisioni associato.



a) schema generale;

b) esempio: nel nodo di decisione *q=1* si sceglie prima il ramo *l=1* che porta a un nodo terminale F. Si torna indietro (backtracking) fino al più vicino nodo di decisione, si prende l'altro ramo(*k=1*) che porta a un nodo di decisione *r=1*. Qui, la prima scelta - *m=1* - porta a un nodo terminale S.

Si consideri ad esempio il seguente caso:

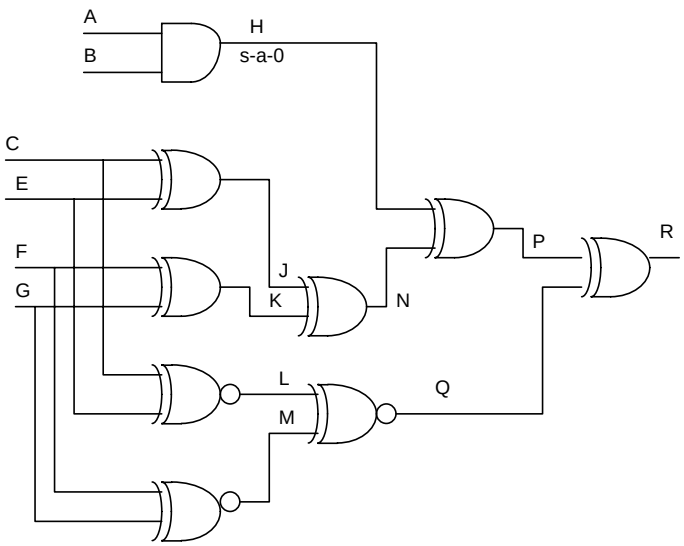


si cerca un test per *f s-a-0*: occorre giustificare un 1 su *f*. Si prova prima *b=0* (ramo di sinistra nell'albero delle decisioni): questo provoca *e=1*, il che impedisce la propagazione dell'errore fino a *h*. Si tenta quindi *c=0*: diventa *g=1*, di nuovo un valore che impedisce la propagazione dell'errore. Non esistono altre possibilità, quindi non esiste un test per il guasto *f s-a-0*, che è ridondante.

Il primo algoritmo basato sulla D-algebra è, come accennato, il D-algorithm di Roth, nel quale si inizia con l'assegnare valori arbitrari (ovunque ci siano possibilità di scelta) a linee *interne* e poi si procede verso gli ingressi primari: può rendersi necessario un *percorso a ritroso* (backtrace) in corrispondenza di ognuna delle porte sul percorso e tentando anche le *sensibilizzazioni di percorsi multipli*. Nel caso pessimo, l'esplorazione completa dell'albero delle decisioni analizzando tutte le alternative sulle linee interne porta a complessità *esponenziale col numero di porte nel circuito*. Il caso ottimo si ha quando non occorre backtracking - allora la complessità è *lineare col numero delle porte*. Una soluzione (in media) più efficiente è stata proposta con l'algoritmo PODEM (Goel, 1980) (Path Oriented DEcision Making), che ricorre a una tecnica di *ricerca diretta*, operando le decisioni solo sugli ingressi primari. Un valore v_k da giustificare sulla linea k è un *obiettivo* (k,v_k) che deve essere raggiunto con un assegnamento sugli ingressi primari: questi valori vengono poi propagati alle linee interne con passi di implicazione. Il backtracking può quindi coinvolgere solo gli ingressi primari, mai le linee interne.

PODEM esamina (implicitamente ma in modo esaustivo) tutti i vettori d'ingresso come possibili vettori di test per un dato guasto; non appena trova che una combinazione è un vettore di test, la ricerca termina. Se nessun vettore d'ingresso è un vettore di test, il guasto viene classificato ridondante.

Si consideri ad esempio un albero di EXOR (caso molto complesso per la generazione del test):



Guasto sotto test: H s-a-0.

Operazioni svolte da PODEM:

1) Si assegna un valore a un ingresso arbitrario: es.: A=1.
2) Si calcola l'implicazione di tutti gli ingressi primari. Il solo ingresso A non è sufficiente a implicare l'uscita di nessuna porta, quindi si assegna un secondo ingresso primario - es.: B=1 - in modo da eccitare il guasto.
3) Si valuta di nuovo l'implicazione degli ingressi primari.A=1 e B=1 implicano H=D (H deve valere 1 se sano, 0 se guasto).
4) L'obiettivo è ora propagare D o $\overline{D}$all'uscita primaria. Per propagare D da H a P, si assegna C=1 ed E=0: questo implica J=1 e L=0. Non è ancora sufficiente per propagare il valore D su H; si assegna F=0 e G=0. Le implicazioni sugli ingressi primari forzano K=0, M=1, N=1, Q=0.
5) H=D e N=1 implicano P=$\overline{D}$; P=$\overline{D}$ e Q=0 implicano R=$\overline{D}$. Si è completata la generazione del test.

La scelta del guasto per cui si vuole determinare il vettore di test e degli ingressi su cui fare, nell'ordine, i vari assegnamenti viene guidata da opportune regole, in modo da ridurre (in media) la necessità di risalire lungo l'albero delle scelte per seguire altre alternative.

Sono state proposte successive modifiche a PODEM per migliorare ulteriormente le prestazioni, in particolare per circuiti dotati di fanout riconvergente.

Le tecniche di test logico possono essere estese (con una certa complessità) a circuiti sequenziali. L'estensione del D-algorithm a circuiti sincroni: prevede la sostituzione di un circuito sequenziale *sincrono* con una *rete iterativa*, dotata di latch per stabilizzare i valori delle uscite laterali: la singola cella (e i relativi latch) rappresentano una "istantanea" del circuito nel corrispondente tempo di clock ("time frame"). I problemi principali derivano dai seguenti fatti:

1. un guasto singolo nel circuito sequenziale viene sostituito con un guasto multiplo nella rete iterativa equivalente;
2. in genere, lo stato iniziale della rete non è noto - quindi non sono noti gli ingressi laterali alla cella sotto test;

il procedimento di generazione dei vettori di test può essere estremamente lungo: si ricorre di solito preferibilmente a tecniche specifiche per il test delle reti sequenziali.

C a p i t o l o

BIBLIOGRAFIA

[1{Bib_GDN93}] A. Ghosh, S. Devadas, A.R. Newton, "Sequential Test Generation and Synthesis for Testability at the Register-Transfer and Logic levels", *IEEE Trans. on CAD*, vol. 12, no 5, pp. 579-598, May 1993.

[2{Bib_VS86}] A. Vergis, K. Steiglitz, "Testability conditions for bilateral arrays of combinational cells", *IEEE Transactions on Computers*, vol. C-35, n.1, pp. 13-22, January 1986.

[3{Bib_Array_comb}] A.D. Friedman, "Easily Testable Iterative Systems", *IEEE Transactions on Computers*, vol. C-22, n.12, pp. 1061-1064, December 1973.

[4{Bib_Dill_1994}] A.J. Hu, G. York, D.L. Dill, "New Techniques for Efficient Verification with Implicitly Conjoined BDDs", *Proc. 31st ACM/IEEE Design Automation Conference – DAC*, pp. 276- 282, 1994.

[5{Bib_Aut}] *Autologic VHDL Reference Manual*, Mentor Graphics, Rel. A3F, 1993.

[6{Bib_Algebra}] B. Garret, M. Saunders, *Algebra*, Ed. London Macmillan, 1971.

[7{Bib_LN89}] B. Lin, A. R. Newton, "Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages" *IFIP International Conference on VLSI*, pp. 187-196, August 1989.

[8{Bib_LJK95}] B. Lin, G. de Jong, T. Kolks, "Modeling and Optimization of Hierarchical Synchronous Circuits" *EDTC-95: IEEE European Design and Test Conference*, pp. 144-149, Paris, France, March 1995.

[9{Bib_Castor}] C. Bolchini, G. Buonanno, F. Ferrandi, D. Sciuto, M. Bombana, P. Cavalloro, "A Wafer Level Testability Approach Based on an Improved Scan Insertion Technique", *IEEE Transactions on Components, Packaging, and Manufacturing Technology*–Part B, vol. 18, no. 3, pp. 438-447, August 1995.

[10{Bib_BEHATE}] C. Bolchini, G. Buonanno, F. Ferrandi, F. Fummi, D. Sciuto, M. Bombana, P. Cavalloro, P.M. Borrego, "Definition of the methodology for testability analysis at the RTL and CDFG levels. Requirement specs for Functional Pattern Quality Evaluator", Deliverable 2.3.A, ESPRIT Project 20616 – *Reuse and Quality Estimation*, April 1996.

[11{Bib_Armstrong94}] C.H. Cho, J. R. Armstrong, "B-algorithm: A Behavioral Test Generation Algorithm", *Proc. IEEE Int. Test Conf.*, pp. 968-979, 1994.

[12{Bib_sung76}] C.H. Sung, "Testable sequential cellular arrays", *IEEE Transactions on Computers*, vol. C-25, n.1, pp. 11-18, January 1976.

[13{Bib_WC90}]C.-W. Wu, P.R. Cappello, "Easily Testable Iterative Logic Arrays", *IEEE Transactions on Computers*, vol. C-39, n. 5, pp. 640-652, May 1990.

[14{Bib_OPUS}]Chickermane, V. and Patel, J.H., "A Fault Oriented Partial Scan Design Approach", *Proc. IEEE/ACM International Conference on Computer Aided Design – ICCAD*, pp. 400-403, November 1991.

[15{Bib_Gajiski}] D. Gajiski, N.D. Dutt, A.C-H. Wu, S.Y-L. Lin, *High Level Synthesis – Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[16{Bib_HP94}] D.A. Patterson, J.L. Hennessy, *Computer Organization & Design – the Hardware/Software Interface*, Morgan Kaufmann, 1994.

[17{Bib_LR_90}] D.H. Lee, S. M. Reddy, "On Determining Scan Flip-Flops in Partial-Scan Design", *Proc. IEEE/ACM International Conference on Computer Aided Design – ICCAD*, pp.322-325, November 1990.

[18{ Bib_Per93Bib_perry}] D.L. Perry, *VHDL*, McGraw-Hill, Inc., 1991.

[19{Bib_Tri80}] E. Trischler, "Incomplete Scan Path with an Automatic Test Generation Methodology", *Proceedings International Test Conference*, 1980, pp. 153-162.

[20{Bib_iscas89}] F. Brglez, D. Bryan, K. Kózmínski, "Combinational Profiles of Sequential Benchmark Circuits" *ISCAS-89: IEEE International Symposium on Circuits and Systems*, pp. 1929-1934, Portland, OR, May 1989.

[21{Bib_FSS95}] F. Fummi, D. Sciuto, M. Serra, "Sequential Logic Minimization Based on Functional Testability", *proc. European Design and Test Conference*, pp.207-211, 1995.

[22{Bib_array}] F. Lombardi, D. Sciuto, "On Functional Testing of Array Processors", *IEEE Transactions on Computers*, vol. C-37, no. 11, pp. 1480-1484, November 1988.

[23{Bib_CUDD}] F. Somenzi, *CUDD: CU Decision Diagram Package Release 1.0.7*, Department of Electrical and Computer Engineering, University of Colorado at Boulder.

[24{Bib_bcs91}] G. Buonanno, C. Costi, D. Sciuto, "Fault Modelling and Test Pattern Generation in the Design of Array Processors" , *Proc. IEEE International Symposium on Circuit and Systems*, pp. 2080-2083, Singapore, 1991.

[25{Bib_Caboldi_1993}] G. Cabodi, P. Camurati, "Exploting cofactoring for efficient FSM symbolic traversal based on the Transition Relation", *Proc. IEEE International Conference on Computer Design – ICCD*, pp. 299-303, October 1993.

[26{Bib_DeMicheli}] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill 1994.

[27{Bib_HS95}] G.D. Hatchtel, F. Somenzi, "Logic Synthesis and Verification Algorithms", KAP - Kluwer Academic Publisher, 1995.

[28{Bib_LEDA}] *GRAPHGEN Implementor's Guide*, LEDA, Meylan (France), March 1996.

[29{Bib_CHS93}] H. Cho, G. D. Hachtel, F. Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Transactions on CAD/ICAS*, Vol.CAD-12, No.7, pp.935-945, July 1993.

[30{Bib_EVK86}] H. Elhuni, A. Vergis, L. Kinney, "C-Testability of Two-Dimentional Iterative Arrays", *IEEE Transactions on Computer Aided Design*, vol. CAD-5, n.4, pp. 573-581, October 1986.

[31{Bib_Fuji_1993}] H. Fujii, G. Ootomo, C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams", *Proc. IEEE/ACM International Conference on Computer Aided Design – ICCAD*, pp.38-41, November 1993.

[32{Bib_Fuji85}]H. Fujiwara, "FAN: A Fanout-Oriented Test Pattern Generation Algorithm," *ISCAS-85*, pp. 671-674, Kyoto, Japan, July 1985.

[33{Bib_LL92}] H. T. Liaw, C. S. Lin, "On the OBDD-Representation of General Boolean Functions", *IEEE Transactions on Computers*, vol. 41, n.6, pp. 661-664, June 1992.

[34{Bib_TSL90}] H. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli, "Implicit Enumeration of Finite State Machine using BDDs", *ICCAD*, pp.130-133, 1990.

[35{Bib_WB95}]H. Y. Wang, R. K. Brayton, "Exploitation of Input Don't Care Sequences in Logic Optimization of FSM Networks," *ICCAD-95*, San Jose, CA, Nov. 1995.

[36{Bib_WB93}]H. Y. Wang, R. K. Brayton, "Input Don't Care Sequences in FSM Networks" *ICCAD-93: IEEE/ACM International Conference on Computer Aided Design*, pp. 321-328, Santa Clara, CA, November 1993.

[37{Bib_WB94}]H. Y. Wang, R. K. Brayton, "Permissible Observability Relations in FSM Networks" *DAC-31: ACM/IEEE Design Automation Conference*, pp. 677-683, Anaheim, CA, June 1994.

[38{Bib_PR92}] I. Pomeranz, S. M. Reddy, "The Multiple Observation Time Test Strategy," *IEEE Trans. on Computers*, Vol. C-41, No. 5, pp. 627-637, May 1992.

[39{Bib_PR93}] I. Pomeranz, S.M. Reddy, "Classification of Faults in Synchronous Sequential Circuits", *IEEE Trans. on Computers*,vol. C-42, no. 9, pp. 1066-1077, September 1993.

[40{Bib_Weg94b}] I. Wegener, "Comments on ″A Characterization of Binary Decision Diagrams″", *IEEE Transactions on Computers*, vol. 43, n.4, pp. 383-384, April 1994.

[41{Bib_Weg94}] I. Wegener, "The size of Reduced OBDD's and Optimal Read-Once Branching Programs for Almost All Boolean Functions", *IEEE Transactions on Computers*, vol. 43, n.11, pp. 1262-1269, November 1994.

[42{Bib_KB72}] J. Kim, M. M. Newborn, "The Simplification of Sequential Machines with Input Restrictions," *IEEE Trans. on Computers*, Vol. C-21, No. 12, pp. 1440-1443, Dec. 1972.

[43{Bib_DeMan93}] J. Steensma, W. Geurts, F. Catthoor, H. De Man, "Testability Analysis in High Level Data Path Synthesis", *Journal of Electronic Testing: Theory and Applications*, pp. 43-56, 1993.

[44{Bib_SGC93}] J.F. Santucci, A.L. Courbis, N. Giambiasi, "Behavioral Testing of Digital Circuits", *Journal of Microelectronics Systems Integration*, vol. 1, no. 1, pp. 55-77, 1993.

[45{Bib_Rho94b_ICCAD}] J.K. Rho, F. Somenzi, "Don't Care Sequences and the Optimization of Interacting Finite State Machines", *IEEE Trans. on CAD*, Vol. CAD-13, No. 7, pp. 865-874, July1994.

[46{Bib_BRB90}] K. S. Brace, R. L. Rudell and R. E. Bryant, "Efficient Implementation of a BDD Package", *27th DAC*, pp. 40-45, 1990.

[47{Bib_Che91b}] K. T. Cheng, "An ATPG-Based Approach to Sequential Logic Optimization," *ITC'91: IEEE International Test Conference*, pp.372-375, Nashville, TN, October 1991.

[48{Bib_CDC}] K.A. Bartlett, R.K. Brayton, G. D. Hachtel, R.M. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Multilevel Logic Minimization Using Implicit don't cares", *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 723-739, June 1988.

[49{Bib_KK95}] K.S. Kim, C.R. Kime, "Partial Scan Flip-Flop Selection by Use of Empirical Testability", *Journal of Electronic Testing: Theory and Applications, Special Issue on Partial Scan Methods*, pp. 47-59, 1995.

[50{Bib_Che91}]K.T. Cheng, "An ATPG-based Approach to Sequential Logic Optimization", *Proc. IEEE Int. Test Conf*, pp. 372-375, 1991.

[51{Bib_CA_90}] K.T. Cheng, V. D. Agrawal, "A Partial Scan Method for Sequential Circuits with Feedback", *IEEE Transactions on Computers*, vol. C-39, n.4, pp. 544-548, April 1990.

[52{Bib_EC93}] L. Entrena, K. T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal" *ICCAD-93: IEEE/ACM International Conference on Computer Aided Design*, pp. 310-315, Santa Clara, CA, November 1993.

[53{Bib_SG94}] L. Vandeventer, J.F. Santucci, “Algorithms for Behavioral Test Pattern Generation from VHDL Circuit Descriptions Containing Loop Language Constructs”, *Euro-VHDL ’94*.

[54{Bib_Abramovici Bib_ABF90}] M. Abramovici, M.A. Breuer, A.D. Friedman, *Testing and Testable Design*, Computer Science Press, New York, 1995.

[55{Bib_BBC94}] M. Bombana, G. Buonanno, P. Cavalloro, F. Ferrandi, D. Sciuto, G. Zaza, “ALADIN: A Multilevel Testability Analyzer for VLSI System Design”, *IEEE Transactions on VLSI Systems*, vol. 2, no. 2, pp.157-171, June 94.

[56{Bib_Sequenziali}] M. Bombana, G. Buonanno, P. Cavalloro, F. Ferrandi, D. Sciuto, G. Zaza, “Cycles Analysis for Testability of WSI Sequential Architectures”, *Proc. IEEE International Conference on Wafer-Scale Integration – WSI*, S.

[57{Bib_bcz91}] M. Bombana, P. Cavalloro, G. Zaza, “Italtel SIT Example”, *Technical report 5020/2/1991* - ESPRIT Project 5020, January 1991.

[58{Bib_Dam94}] M. Damiani, “Non-Deterministic Finite State Machines and Sequential Don't Cares” *EDAC-94: IEEE European Conference on Design Automation*, pp. 192-198, Paris, France, March 1994.

[59{Bib_DDM93}] M. Damiani, G. De Micheli, “Don’t Care Set specification in Combinational and Synchronous Logic Circuits”, *IEEE Transactions on Computer-Aided Design*, vol.12, pp. 365-388, March 1993.

[60{Bib_Fujita_1993}] M. Fujita, H. Fujisawa, and Y. Matsunaga, “Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation”, *IEEE Transactions on Computer-Aided Design*, vol.12, pp. 6-12, January 1993.

[61{Bib_RDP95}] M. Potkonjak, S. Dey, R. K. Roy, “Considering Testability at Behavioral Level: Use of Transformations for Partial Scan Cost Minization Under Timing and Area Constraints”, *IEEE Tr. On CAD*, vol. 14, no. 5, pp. 531-546, May 1995.

[62{Bib_PDR_95}] M. Potkonjak, S. Dey, R. K. Roy, “Considering Testability at Behavioral Level: Use of Transformations for Partial Scan Cost Minimization Under Timing and Area Constraints”, *IEEE Transactions on Computer-Aided Design*, vol.14, no. 5, pp. 531-546, May 1995.

[63{Bib_RS59}] M. Rabin, D. Scott, “Finite Automata and Their Decision Problems” *IBM Journal of Research and Development*, pp. 114-125, 1959.

[64{Bib_Sami}] M. Sami, R. Negrini, “Calcolatori Elettronici”, CLUP, 1986.

[65{Bib_HH_ITC95}] M.C. Hansen, J. P. Hayes, “High-Level Test Generation Using Symbolic Scheduling”, *Proc. IEEE Int. Test Conf*, pp. 586-595, 1995.

[66{Bib_HH_VTS95}] M.C. Hansen, J. P. Hayes, “High-Level Test Generation Using physically-induced faults”, *Proc. VLSI Test Symp.*, pp. 20-28, 1995.

[67{Bib_armstrong89}] M.D.O’Neil, D.D.Jani, C.H. Cho, and J.R. Armstrong, “BTG: A Behavioral Test Generator”, *Proc. 9th Int. Symp. On CHDLs*, pp. 347-361, Jun. 1989.

[68{Bib_Abadir_Breuer1}] M.S. Abadir, M.A. Breuer, “A Knowledge-based system for design testable VLSI chips”, *IEEE Design & Test*, pp. 56-68, August 1985.

[69{Bib_Abadir_Breuer2}] M.S. Abadir, M.A. Breuer, “Test schedules for VLSI circuits having built-in test hardware”, *IEEE Transactions on Computers*, vol. C-35, pp. 361-367, April 1985.

[70{Bib_SG93}] N. Giambiasi, J.F. Santucci, A.L. Courbis, “Test Pattern Generation for Behavioral Descriptions in VHDL”, *Euro-VHDL ’91*, pp. 228-235.

[71{Bib_CBM89}] O. Coudert, C. Berthet, J.C. Madre, "Verification of Sequential Machines Using Boolean Functional Vectors", *Proc. IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, L.J.M. Claesen (ed.), North Holland, 1989.

[72{Bib_traversal}] O. Coudert, C. Berthet, J.C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution", in *Automatic Verification Methods for Finite State Systems*, J. Sifakis (ed.), Lecture Notes in Computer Science, vol. 407, Springer-Verlang, 1989.

[73{Bib_CM92}]O. Coudert, J. C. Madre, "Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions," *DAC-29: ACM/IEEE Design Automation Conference*, pp.36-39, Anaheim, CA, June1992.

[74{Bib_Goel81}] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*}, Vol. C-30, No. 3, pp. 215-222, March 1981.

[75{Bib_G80}] P. Goel, "Test Generation Costs Analysis and Projections", *DAC-17: ACM/IEEE Design Automation Conference*, pp.77-84, 1980.

[76{Bib_VAS93}] P. Vishakantaiah, J. A. Abraham, D. G. Saab, "CHEETA: Composition of Hierarchical Sequential Tests Using ATKET," *ITC'93: IEEE International Test Conference*, pp.606-615, 1993.

[77{Bib_VAA92}] P. Vishakantaiah, J. Abraham, M. Abadir, "Automatic Test Knowledge Extraction from VHDL ATKET," *DAC-29: ACM/IEEE Design Automation Conference*, pp.273-278, Anaheim, CA, June 1992.

[78{Bib_PA93}] P.S. Parikh, M. Abramovici, "A Cost Based Approach to Partial Scan", *Proceedings 30th Design Automation Conference*, June 1993.

[79{Bib_PA95}] P.S. Parikh, M. Abramovici, "Testability-Based Partial Scan Analysis", *Journal of Electronic Testing: Theory and Applications, Special Issue on Partial Scan Methods*, pp. 61-70, 1995.

[80{Bib_R93}] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", *Proc. IEEE/ACM International Conference on Computer Aided Design – ICCAD*, pp. 42-47 November 1993.

[81{Bib_Bry86}] R.E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, vol. C-35, No.8, pp.677-691, August 1986.

[82{Bib_Thomas94}] R.S. Ramchandani, D.E. Thomas, "Behavioral Test Generation using Mixed Integer Non-linear Programming", *Proc. IEEE Int. Test Conf.*, pp. 958-967, 1994.

[83{Bib_C93}] S. Chakravarty, "A Characterization of Binary Decision Diagrams", *IEEE Transactions on Computers*, vol. 42, pp. 129-137, February 1993.

[84{Bib_D91}] S. Devadas, "Optimizing Interacting Finite State Machines Using Sequential Don't Cares" *IEEE Transactions on Computer Aided Design*, Vol. CAD-10, No. 12, pp. 1473-1484, December 1991.

[85{Bib_DGK94}] S. Devadas, A. Ghosh, K. Keutzer, *Logic Synthesis*, McGraw-Hill, 1994.

[86{Bib_Freeman}] S. Freeman, "Test Generation for Data-Path Logic: The *F-path* method", *IEEE Journal of Solid-State Circuits*, vol. 23, pp. 421-427, April 1988.

[87{Bib_GC91}] S. Ghosh, T. J. Chakraborty, "On Behavior Fault Modeling for Digital Design", *Journal of Electronic Testing: Theory and Applications*, 2, pp. 135-151, 1991.

[88{Bib_U69}] S. H. Unger, *Asynchronous Sequential Switching Circuits*, John Wiley, New York, NY, 1969.

[89{Bib_Malik_1988}] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", *Proc. IEEE/ACM International Conference on Computer Aided Design – ICCAD*, pp. 6-8, November 1988.

[90{Bib_sbdd}] S. Minato, N. Ishiura, S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", *27th DAC*, pp. 52-57, 1990.

[91{Bib_PS95}] S. Panda, F. Somenzi, "Who are the variables in your neighborhood", *Proc. International Conference on Computer Aided Design*, pp. 74-77, San Jose, CA, November 1995.

[92{Bib_PSP94}] S. Panda, F. Somenzi, B.F. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams", *Proc. International Conference on Computer Aided Design*, pp. 628-631, San Jose, CA, November 1994.

[93{Bib_mcnc91}] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0" *MCNC Technical Report*, Research Triangle Park, NC, January 1991.

[94{Bib_Ake78}] S.B. Akers, "Binary Decision Diagrams", *IEEE Trans. on Computers*, vol. C-27, no.6, pp.509-516, June 1978.

[95{Bib_CBA95}] S.T. Chakradhar, A. Balakrishnan, V.D. Agrawal, "An Exact Algorithm for Selecting Partial Scan Flip-Flops", *Journal of Electronic Testing: Theory and Applications, Special Issue on Partial Scan Methods*, pp.83-91, 1995.

[96{Bib_kung_88}] S.Y. Kung, "VLSI Array Processor", *Prentice Hall*, Englewood Cliffs, New Jersey, 1988.

[97{Bib_Syn}] *Synopsys User's Manual*, Synopsys Inc., 1994.

[98{Bib_Synthesia}] *SYNTH 1.5 User's Guide*, Synthesia AB, Kista (SE), 1995.

[99{Bib_KVB95}] T. Kam, T. Villa, R. K. Brayton, A. Sangiovanni-Vincentelli, "Implicit State Minimization of Non-Deterministic FSMs" *IWLS-95: IEEE International Workshop on Logic Synthesis*, Lake Tahoe, CA, May 1995.

[100{Bib_Lin84}] T. Lin, S. Y. H. Su, "The S-algorithm: A promising Solution for Systematic Functional Test Generation" *IEEE Tr. On CAD*, vol. 4, pp. 250-263, July 1985.

[101{Bib_NP91}] T. Nierman, J.H. Patel, "HITEC: a test generation package for sequential circuits", *EDAC*, pp.214-218, 1991.

[102{Bib_OKP95}] T. Orenstein, Z. Kohavi, I. Pomeranz, "An Optimal Algorithm for Cycle Breaking in Directed Graphs", *Journal of Electronic Testing: Theory and Applications, Special Issue on Partial Scan Methods*, pp.71-81, 1995.

[103{Bib_CB85}] T.H. Chen, M.A. Breuer, "Automatic Design for Testability Via Testability Measures", *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, pp.3-11, January 1985.

[104{Bib_GC95}] U. Glaser, K. T. Cheng, "Logic Optimization by an Improved Sequential Redundancy Addition and Removal Technique," *IEEE Asia South-Pacific Design Automation Conference*, pp.235-240, 1995.

[105{Bib_DP89}] U. J. Davé, J. H. Patel, "A Functional-Level Test Generation Methodology Using Two-Level Representations"' *DAC-26: ACM/IEEE 26th Design Automation Conference*, pp. 722-725, Las Vegas, NV, June 1989.

[106{Bib_HLS88}] W-K Huang, F. Lombardi, D. Sciuto, "Design and Analysis of C-testable Arrays", *Wafer Scale Integration II*, R.M. Lea (ed.), NorthHolland, pp. 115-123, 1988.

[107{Bib_GKP94}] X. Gu, K. Kuchcinski, Z. Peng, “Testability Analysis and Improvement from VHDL Behavioral Specifications,” *EuroVHDL’94: IEEE European VHDL Conference*, pp.644-649, Grenoble, France, September 1994.

[108{Bib_Hsu95}] Y. C. Hsu etc., *VHDL Modeling for Digital Design Synthesis*, Kluwer Academic Publishers, 1995.

[109{Bib_WatB94}] Y. Watanabe, R. K. Brayton, “State Minimization of Pseudo Non-Deterministic FSMs” *EDAC-94: IEEE European Conference on Design Automation*, pp. 184-191, Paris, France, March 1994.

[110{Bib_WatB93}] Y. Watanabe, R. K. Brayton, “The Maximum Set of Permissible Behaviors for FSM Networks” *ICCAD-93: IEEE/ACM International Conference on Computer Aided Design*, pp. 316-320, Santa Clara, CA, November 1993.

[111{Bib_Benchmarks}{Bib_Bench92}] 1991 and 1992 High Level Synthesis benchmarks, retrieved via anonymous ftp at *mcnc.mcnc.org* in */pub/benchmark/HLSynth91 [92]*.