Appunti di Progettazione Hardware 2

Matteo Gianello

12 luglio 2014

Indice

1 Test di circuiti digitali

Quando si vuole realizzare un nuovo prodotto hardware come uno smartphone o semplicemente anche un nuovo modulo wifi si parte sempre dalle richieste del cliente e da queste si passa alla stesura di un documento di *specifica*, alla *progettazione* del sistema ed infine alla *verifica* che il sistema progettato è conforme alla specifica. Dopo queste fasi iniziali però si deve tradurre il progetto in uno hardware che si possa costruire in modo semplice ed efficiente. Il passo finale prima di poter commercializzare il prodotto è quello del collaudo in quanto, in molti casi possono sorgere problemi di fabbricazione e, pur essendo il progetto corretto il sistema finale non funzionerà. Lo scopo di questo capitolo allora è quello di capire quali sono le tecniche migliori per far collaudare un nuovo sistema. Primo fra tutti è quello di pensare al collaudo durante la fase di progettazione e prevedere un modo per rendere facile i test dopo aver prodotto il sistema. Il modo più semplice per testare un circuito è quello di prelevare il componente terminato e inserire una serie di valori in ingresso e comparare i valori all'uscita per verificare che il risultato sia corretto come mostrato in Figura 1.

Questo meccanismo può essere applicato a qualsiasi componente del circuito ma questa tecnica presenta diversi problemi, prima tra tutte il costo delle apparecchiature necessarie per l'esecuzione dei test; ad esempio per un ATE a 1024 pin che lavora ad una frequenza che varia tra *0.5* e *1 GHz* il costo è dato dal costo base più un certo costo per ogni pin.

$$\$1.2M + 1024 \times \$3000 = \$4,27M$$

Per tale apparecchiatura si dovrà inoltre prevedere un costo dovuto all'ammortamento, alla manutenzione e alla sua operatività che per il nostro esempio possiamo quantificare come:

$$\begin{aligned} &= Ammortamento + Manutenzione + Operativita \\ &= \$0.854M + \$0.085M + \$0.5M \\ &= \$1.439M/anno \end{aligned}$$

Tale costo si riflette sul prodotto finale, infatti, supponendo che la macchina lavori 24/7 avvremmo che il costo al secondo della macchina è pari a:

$$\begin{aligned} &= \$1.439M/(365 \times 24 \times 3600) \\ &= 4.5cent/sec \end{aligned}$$

Questo significa che ogni secondo di test costa *4.5 cent* per questo motivo è necessario che il test dia risultati chiari ma che venga svolto nel minor tempo possibile.

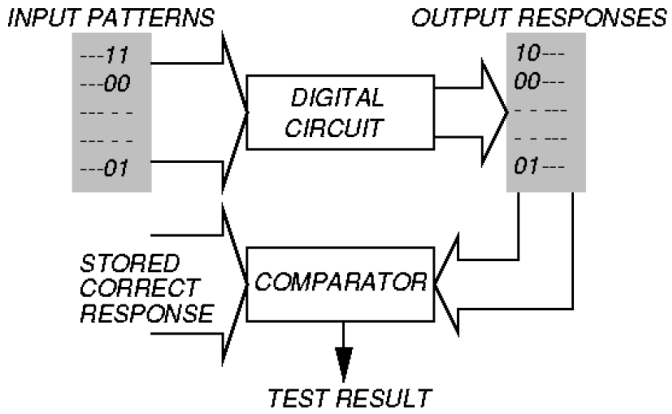


Figura 1: Collaudo di un sistema hardware

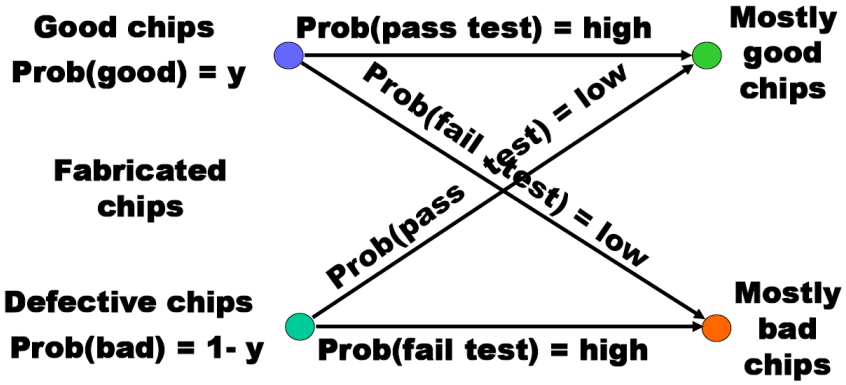


Figura 2: Probabilità di individuazione di componenti guasti

Un altro problema che si può presentare è la granularità con la quale si effettuano i test, infatti più il livello al quale si effettua il test è basso meno costoso sarà la riparazione, ad esempio se il livello al quale si scopre il guasto è a livello di transistor il costo della sostituzione del transistor è quantificabile a *1*, se il guasto viene individuato quando il transistor è montato su di una board allora il costo sale a *10* se tale board è montata nel sistema prima di scoprire il guasto allora il costo di quel guasto sale a *100* e così via; mano a mano che si sale di livello il costo di un guasto in termini di individuazione e riparazione diventa dieci volte più grande. Per questo motivo quando si effettuano i test le probabilità di individuare un componente difettoso devono essere le più alte possibili anche a costo di scartare qualche componente che non sia difettoso come mostrato in Figura 2.

1.1 I modelli di guasto

L'individuazione dei difetti di fabbricazione è diventata perciò una parte fondamentale della fase di realizzazione di un sistema hardware, e come tale si è evoluta nel tempo passando da considerare i *difetti di fabbricazione* come eventi sfortunati, a considerarli successivamente dei *guasti* fino ad arrivare ai *modelli di guasto* che si prefiggono lo scopo di creare un modello per ogni livello di astrazione per individuare i punti cruciali del circuito nel quale effettuare i test.
I guasti possono essere di diverso tipo e dovuti a diversi fattori, essi possono essere:

- Difetti di processo:
  - Contatti mancanti
  - Capacità parassite
  - Ossidazione dei contatti
- Difetti materiali:
  - Difetti di carico (rotture o imperfezione nei cristalli)
  - Impurità sulle superfici
- Guasti dipendenti dal tempo:
  - Rotture elettriche
  - Elettromigrazione

Defect classes	Occurrence frequency (%)
Shorts	**51**
Opens	**1**
Missing components	**6**
Wrong components	**13**
Reversed components	**6**
Bent leads	**8**
Analog specifications	**5**
Digital logic	**5**
Performance (timing)	**5**

Figura 3: Percentuale dei guasti ad una PCB

- Guasti da imballaggio

In Figura 3 sono mostrati i principali guasti che si presentano su una PCB con le rispettive percentuali.
Esistono diversi modelli di guasto i più comuni ed utilizzati sono:

- Single stuck-at faults
- Transistor open and short faults
- Memory faults
- PLA faults
- Functional faults
- Delay faults
- Analog faults

1.1.1 Single Stuck-at fault

Il primo modello che analizziamo è il *Single stuck-at* in questo tipo di modello si assume che una linea del circuito rimanga bloccata ad un livello sia esso alto o basso. Il *single stuck-at* ha tre caratteristiche che lo contraddistinguono, la prima è che solamente una linea si guasta, la seconda è che la linea guasta si blocca permanentemente ad un valore *0* o *1*, ed infine, la linea che si blocca può essere sia all'ingresso che all'uscita di una porta. Un esempio di circuito XOR analizzato con modello di guasto a *single stuck-at* è mostrato in Figura 4 dove sono indicati con un pallino verde i 12 punti di guasto mentre i possibili guasti sono 24 (due valori per ogni punto). Questo modello seppur semplice è supportato dall'evidenza empirica infatti è semplice da usare, può individuare anche altri tipi di guasti ed inoltre si presta ad essere modellizzato.
Questo modello può essere ampliato tramite la nozione di *equivalenza di guasto* e quella di *collasso dei guasti*. Si ha un'*equivalenza tra guasti* quando due guasti f_1 e f_2 se tutti i test che individuano il guasto f_1 individuano anche il guasto f_2, se due guasti sono equivalenti le loro funzioni di guasto sono *identiche*. Il *collasso di guasti* si ha quando un circuito logico può essere diviso in un sottoinsieme equivalente dove tutti i guasti del sottoinsieme sono equivalenti, il collasso dei guasti avviene prelevando un guasto da ogni sottoinsieme. In Figura 5 sono mostrate le regole di equivalenza per le diverse porte logiche.

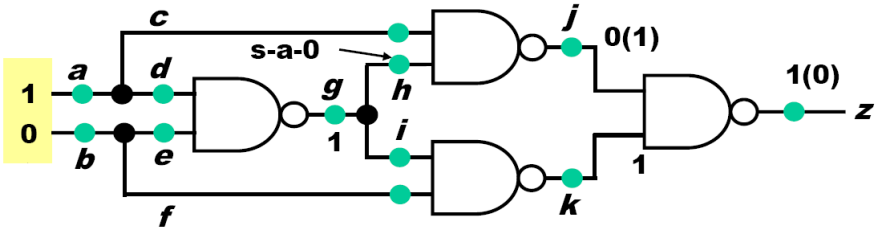


Figura 4: Esempio di modello di guasto *single stuck-at*

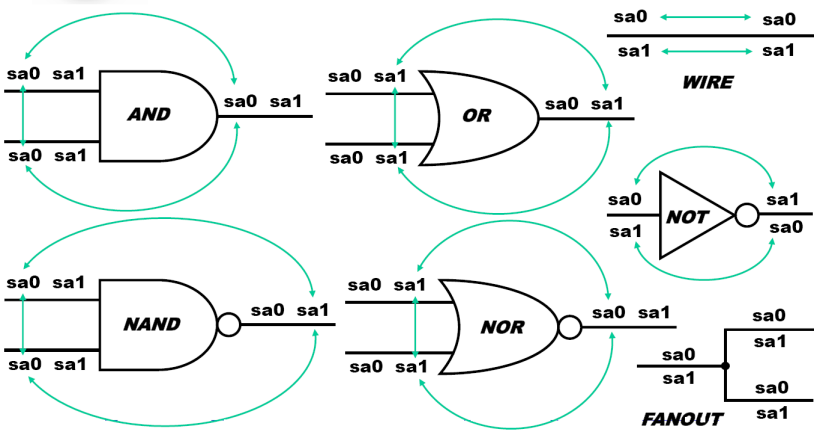


Figura 5: Equivalenze tra le diverse porte logiche

Un altro concetto che vogliamo introdurre è quello della *dominanza* tra guasti, se tutti i test che individuano f_1 individuano anche un secondo guasto f_2 allora si dice che f_2 domina f_1.
Gli ingressi primari e i bracci dei *fanout* di un circuito combinatorio sono chiamati *checkpoints*; un teorema afferma che se un insieme di test individua i guasti su tutti i checkpoint di un circuito combinatorio allora quello stesso insieme di test individua tutti i guasti del circuito.
Il modello a *multiple stuck-at* prevede, rispetto al modello a singolo stuck-at, che più linee contemporaneamente possano essere bloccate in una determinata combinazione di guasto. Il numero totale di combinazioni di guasti in un circuito con k punti di guasto è di $3^k - 1$.

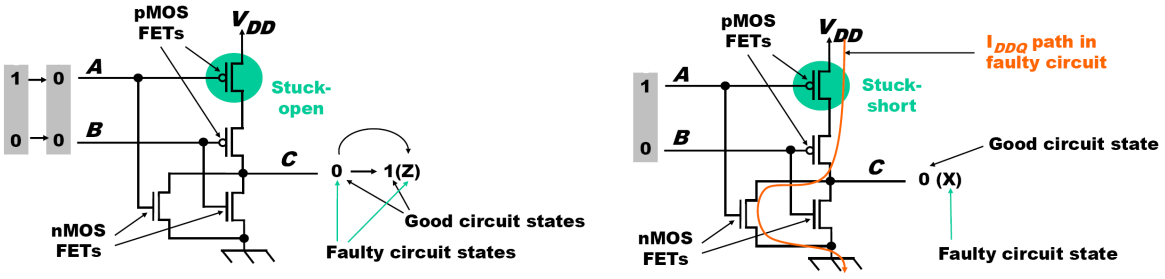
1.1.2 Transistor faults

Esistono diversi modelli di guasto che riguardano i transistor che fanno riferimento a dei comportamenti anomali di questi. I modelli che noi vedremo sono:

Stuck-open: nel quale un singolo transistor è modellizzato come un circuito aperto

Stuck-short: nel quale il transistor viene trattato come un cortocircuito

Nel caso dello *stack-open* sono necessari due vettori per individuare il guasto in quanto il sistema non è in grado di cambiare lo stato precedente del circuito perciò per poterlo testare prima bisogna portare lo stato dell'uscita in uno stato noto e poi cercare di invertire questo stato; in caso di guasto questo cambio risulta impossibile. Per quanto riguarda il caso di *stack-short* invece per identificare i guasti è necessario andare a misurale la *corrente di dispersione* I_{DDQ}del circuito in quanto in un circuito stabile nel quale sono esauriti i transitori non dovrebbe esserci un passaggio di corrente; tuttavia con le nuove tecnologie le correnti di quiescienza sono diventate sempre minori e quindi difficili da individuare.



(a) stuk-open (b) stuk-short

Figura 6: Esempio di stuck-open e stuck-short

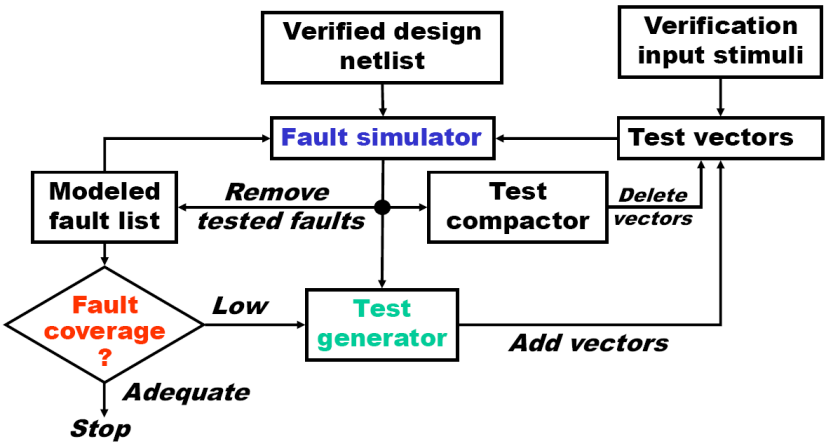


Figura 7: Diagramma che mostra il procedimento e l'utilità della *fault simulator*

1.2 Fault simulation

Dopo aver capito che cos'è un modello di guasto passiamo ora a capire come sfruttarlo all'interno di una simulazione. Per effettuare una simulazione si parte sempre da un circuito, nella maggior parte dei casi questo è espresso a livello logico, da una sequenza di vettori di test e da un modello di guasto; questo per determinare la copertura dei guasti individuati dall'insieme dei vettori di test, ma anche gli eventuali guasti non coperti per migliorare ed ottimizzare i collaudi post-produzione. In alcuni casi la simulazione è utile anche per identificare quelle aree di test che risultano difficili da testare in modo che i progettisti possano valutare la possibilità di aggiungere della logica per aumentarne la testabilità; questo tipo di progettazione è anche chiamata *Design for Test (DFT)*. In Figura 7 vediamo quali sono le diverse utilità della simulazione dei guasti sia in termini di copertura dei guasti (*fault coverage*) sia in termini di ottimizzazione dei test (*test compactor*). Un'altra funzionalità per cui la simulazione dei guasti è utilizzata è quella di *timing* ovvero capire i tempi di transitorio che il circuito ha in presenza di guasti, in quanto potrebbe essere che il circuito ottenga un valore corretto ma che il tempo del transitorio sia maggiore di quello di clock e quindi non riuscirei ad analizzarlo. Questo tipo di simulazione è stata introdotta solamente negli ultimi anni quando le frequenze sono aumentate considerevolmente.
Gli algoritmi per la simulazione dei guasti sono diversi con diversi gradi di difficoltà, queste simulazioni sono:

- Seriale
- Parallela

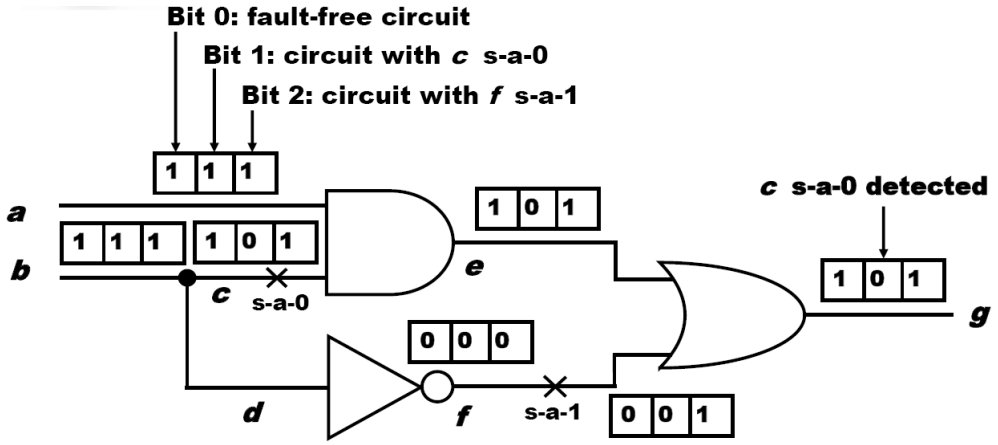


Figura 8: Esempio di simulazione parallela con processore a 3 bit

- Deduttiva
- Concorrente
- e molti altri ancora

1.2.1 Simulazione seriale

Il primo tipo di simulazione che andiamo ad analizzare è quella che sfrutta un algoritmo di tipo *seriale* in quanto è quello più semplice.
Si parte da una simulazione del circuito senza guasti in modo da registrare le uscite del circuito. A questo punto si effettua un'*iniezione del guasto* ovvero si modifica il modello per far si che sia presente il guasto. A questo punto si riapplica il vettore di test al circuito e se la risposta differisce dal caso senza guasto sono in grado di affermare che il vettore di test analizzato identifica il guasto appena testato. Questa operazione viene ripetuta per tutti i guasti nella lista dei guasti, alla fine di tutta la simulazione posso determinare quali guasti sono coperti da quel vettore di test.
Per simulare un guasto durante una simulazione abbiamo detto che dobbiamo fare una *iniezione di guasto* questa tecnica però può essere implementata in due modi, il primo è passare di volta in volta al simulatore un circuito diverso dove è stato introdotto il guasto oppure passare al simulatore un circuito simile a quello da testare ma dove nei punti di guasto sono stati inseriti dei *multiplexer* e selezionare i punti di guasto tramite i valori di controllo del multiplexer.
Lo svantaggio principale di questa tecnica è che richiede un notevole numero di simulazioni e le computazioni risultano ripetitive. Per risolvere tale problema si può sfruttare un certo grado di parallelizzazione.

1.2.2 Simulazione parallela

Nella simulazione parallela si utilizza una caratteristica dei processori, ovvero la possibilità di eseguire un `AND` o un `OR` di un intera parola (32 o 64 bit) in un unico ciclo di clock. L'idea è quella di compattare più di un simulazione durante un'unica computazione, sfruttando tale tecnica si possono testare $w - 1$ (è necessario simulare anche il circuito senza guasti) guasti in un unica simulazione. Un esempio di questa tecnica è mostrato in Figura 8 dove si fa l'esempio di un processore a 3 bit e si testano due guasti con un'unica simulazione.

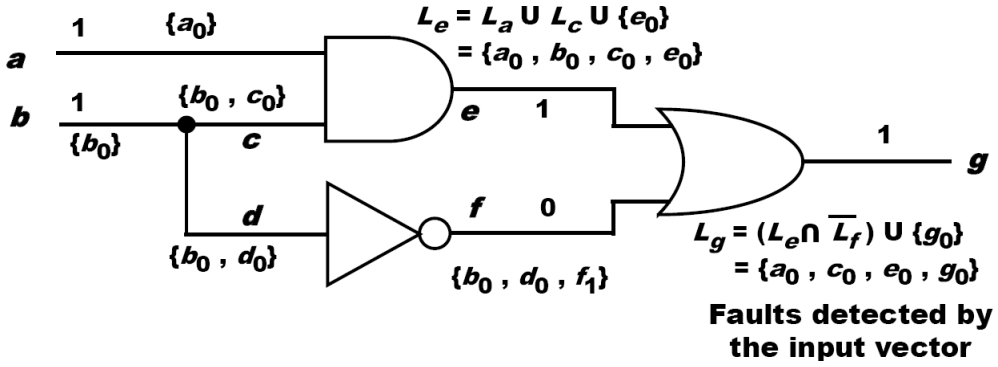


Figura 9: Esempio di simulazione deduttiva

1.2.3 S§imulazione deduttiva

Nel caso di simulazione parallela si sfrutta la possibilità di parallelizzare i calcoli, tuttavia non si agisce sul fatto di riutilizzare le precedenti computazioni per quelle parti di circuito che non vengono toccate da guasti. La simulazione deduttiva, invece, sfrutta proprio questa caratteristica. In questo caso si analizza il circuito con la presenza di tutti i guasti ma considerando un guasto alla volta. Tramite questa tecnica si riesce a calcolare l'insieme dei guasti individuabili tramite un determinato vettore di test utilizzando un'unica simulazione.
Il principio è quello di individuare tutti i possibili guasti di una determinata linea k e inserirli in una lista L_k e a questo punto applicare delle regole insiemistiche delle varie porte. La lista dei guasti che si ottengono all'uscita è l'insieme dei guasti individuabili dal vettore di test utilizzato. Lo svantaggio di questa tecnica è che si applica solamente a circuiti logici booleani e non a quelli con *flip-flop* o altri componenti. In Figura 9 vediamo un esempio di questa tecnica, all'uscita della porta `AND` abbiamo che l'insieme dei guasti rilevabili sono :

$$L_e = L_a \cup L_c \cup \{e_0\} = \{a_0, b_0, c_0, e_0\}$$

Questo perchè entrambe le linee di ingresso mostrano valori non dominanti nel caso invece della porta `OR` abbiamo che all'uscita i guasti individuati sono quelli dell'ingresso e meno quelli dell'ingresso f in quanto sull'ingresso f è presente un valore non dominante.

1.2.4 Simulazione concorrente

La simulazione concorrente è molto simile a quella deduttiva, tuttavia in questo caso si parte da una simulazione priva di guasti e via via si iniettano i diversi guasti ricalcolando solamente le porte che subiscono una variazione dopo l'iniezione. Questo sistema è molto più rapido dei precedenti, tuttavia il fatto di memorizzare lo stato delle porte che cambiano comporta un grande consumo di memoria.
In Figura 10 vediamo un esempio di tale tecnica, si nota subito come l'ultima porta or debba tener conto di tutti i guast.

1.2.5 Simulazione di vettori parallela

Questo tipo di simulazione è molto simile a quella parallela ma in questo caso non si testano diversi guasti con lo stesso vettore bensì si testa un guasto diversi vettori di test. Questa tecnica è utile soltanto se il circuito è privo di memoria. In caso contrario tale tecnica è inapplicabile.

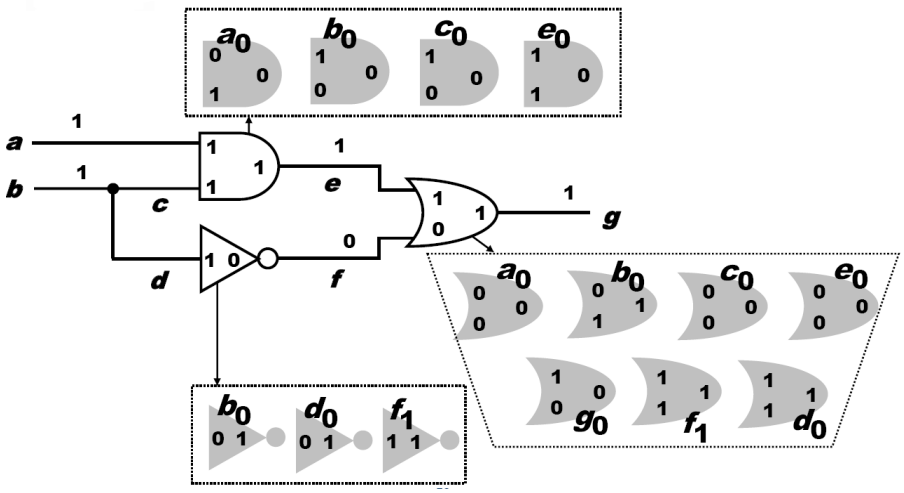


Figura 10: Esempio di simulazione concorrente

1.2.6 Fault samplig

In moti casi è impossibile testare tutti i guasti, tuttavia è possibile ottenere una buona copertura utilizzando il *fault samplig*. Si tratta di scegliere un sottoinsieme casuale dei guasti ed effettuare la simulazione su questi guasti per calcolarne la copertura. La misura di copertura dei guasti su questo sottoinsieme ci da una misura sulla copertura totale dei guasti. Questo meccanismo permette di risparmiare molto tempo nelle simulazioni tuttavia può portare a non testare alcuni guasti.

1.3 Generazione dei test nei circuiti combinatorio

In questo paragrafo analizzeremo come creare i vettori di test che serviranno poi sia per la simulazione che per il collaudo del sistema. In molti casi questi vettori sono generalmente gli stessi che il progettista usa per verificare la corretta funzionalità del sistema rispetto alle specifiche. Tuttavia in molti casi questi vettori non sono sufficienti e a volte necessari per valutare tutti i guasti. Prendiamo ad esempio il caso in Figura 11, in questo caso tenendo conto solo della funzionalità e volendo testare completamente il circuito dovremmo utilizzare 2^{64} valori per il primo ingresso moltiplicati per 2^{64} valori per il secondo ingresso per l'ingresso di riporto per un totale di 2^{129} vettori di test. Nel caso in cui prendiamo in considerazione la struttura del circuito abbiamo che gli stuck-at per la parte di somma di un bit sono 10 mentre per la parte di riporto (non presente in Figura) comprende 17 stuck-at questo significa per ogni singolo bit i guasti da testare sono 27 che moltiplicati per i 64 bit del sommatore fanno un totale di 1728 guasti; considerando il caso peggiore in cui sia necessario un vettore di test per ogni guasto abbiamo che sono necessari $1728 << 2^{129}$ vettori.

Come abbiamo visto conoscere la struttura del circuito ci aiuta a ridurre i vettori di test, un altro fattore che aiuta nel ridurre la complessità delle simulazioni è l'utilizzo di un algoritmo per la generazione dei test che sia *completo* ovvero per ogni guasto l'algoritmo è in grado di stabilire se questo è testabile o meno.

Un guasto *non testabile* è un guasto che, pur essendo presente nel circuito, non influenza il comportamento di tale circuito ovvero non ne compromette la funzionalità, questo tipo di guasto è anche detto *ridondante* ma solo nei circuiti combinatori e indica la presenza di hardware non necessario.

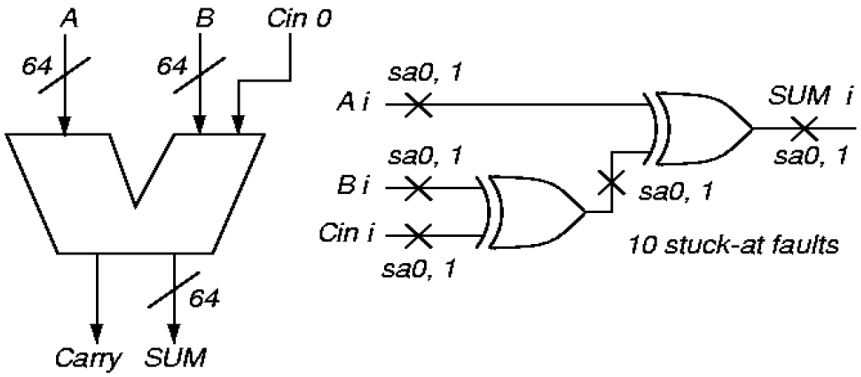


Figura 11: Schema funzionale e strutturale di un full-adder con riporto

Symbol	Meaning	Good Machine	Failing Machine	
D	$1/0$	1	0	
$\overline{D}$	$0/1$	0	1	
0	$0/0$	0	0	Algebra di Roth
1	$1/1$	1	1	
X	X/X	X	X	
$G0$	$0/X$	0	X	
$G1$	$1/X$	1	X	Algebra di Muth
$F0$	$X/0$	X	0	
$F1$	$X/1$	X	1	

Tabella 1: Notazioni delle algebre di Roth e di Muth

Per lo studio della generazione di test dobbiamo prima introdurre alcuni strumenti tra questi le algebre di *Roth* e di *Muth* espresse in Tabella 1. Queste algebre permettono di simulare due macchine simultaneamente, il primo valore indica una macchina che funziona correttamente il secondo valore una macchina in errore, inoltre queste algebre sono di facile rappresentazione e risoluzione. Per trovare i vettori di test dobbiamo però fare delle premesse, il valore nel punto di guasto deve essere differente rispetto a quello dello stuck-at (deve essere presente una D o una $\overline{D}$), il guasto deve essere propagato all'uscita (una D o una $\overline{D}$ devono essere presenti sull'uscita). Analizziamo ora alcuni dei principali algoritmi di generazione dei test.

1.3.1 Algoritmo esaustivo

Nel caso di un circuito con n input si generano tutti i 2^n possibili vettori di test. Questo algoritmo è applicabile solo per un numero $n \leq 15$ di input altrimenti il numero di vettori risulta troppo grande. Tuttavia questo meccanismo può essere utilizzato su alcuni circuiti per effettuare dei test *built-in* su una parte del circuito.

1.3.2 Generazione casuale

In questo caso i vettori di test sono generati casualmente. Questo meccanismo è utile soprattutto nel caso di circuiti aritmetici e fornisce dei risultati buoni con una copertura dei guasti che si aggira tra il 60% e l'80%. Questa tecnica è utilizzata anche in combinazione con altre tecniche per aumentare il grado di copertura.

1.3.3 Path Sensitization

La *Path sensitization* è un algoritmo per l'individuazione dei vettori di test molto semplice e di facile applicazione. Esso si compone di tre passi:

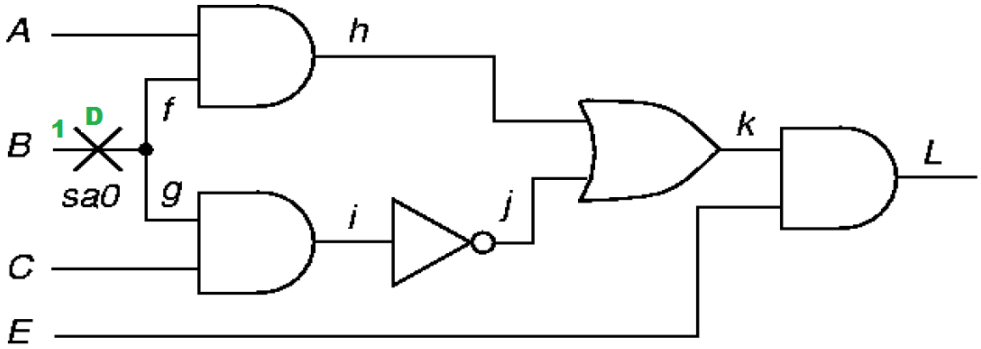
1. **Fault Sensitization:** durante questo passo si innesta il guasto e a questo punto tramite l'algebra di Roth si inserisce una D o una $\overline{D}$ nel punto di guasto (Figura 12(a))

2. **Fault Propagation:** durante questo passo si propaga il guasto verso l'uscita scegliendo uno o più percorsi come mostrato in Figura 12(b) e Figura 12(c)

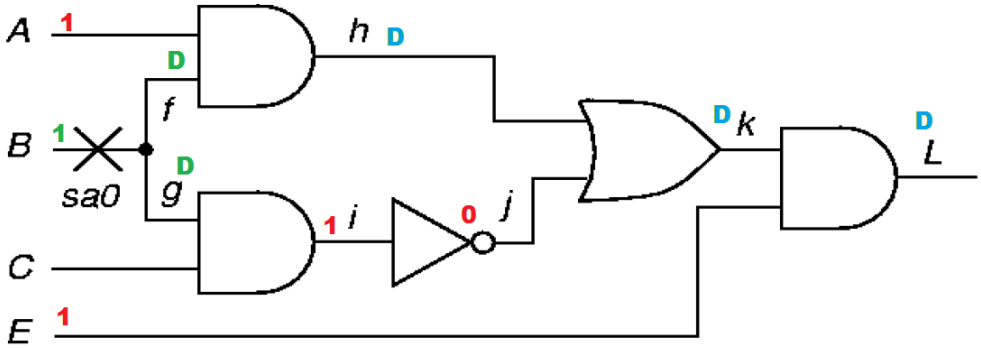
3. **Line Justification:** a questo punto si impongono sulle linee restanti dei valori utili per completare il circuito. Nel caso in cui sia possibile (Figura 12(d)) ottengo il vettore di test, nel caso non sia possibile (Figura 12(b) e 12(c)) scarto il circuito

Analizziamo più in dettaglio l'esempio, in Figura 12(a) vediamo come venga iniettato un guasto con stuck-at 0 ciò significa che l'ingresso B deve essere inizializzato a *1* altrimenti sarebbe impossibile individuare il guasto, inoltre secondo l'algebra di Roth un passio da $1 \leftarrow 0$ va indicato da una D.
In Figura 12(b) il guasto viene propagato lungo il percorso *f-h-k-L* tale propagazione è indicata in blu, dopo aver propagato il guasto fino all'uscita abbiamo cercato di andare a ritroso per riempire le linee rimaste vuote (numeri in rosso). In questo caso, per la porta `AND` con ingressi k ed E ad E abbiamo assegnato il valore *1*, alla porta *AND* con ingressi A e B all'ingresso libero abbiamo assegnato il valore *1*, infine, alla porta `OR` il cui ingresso j era ancora libero abbiamo assegnato il valore *0* e lo abbiamo propagato all'indietro fino ad avere in posizione i il valore *1*, tuttavia, questo valore è incompatibile con qualsiasi valore che avremmo potuto assegnare all'ingresso C; questo porta a scartare il percorso.
In Figura 12(c) il guasto viene propagato parallelamente sia lungo il percorso *f-h-k-L* sia lungo il percorso *g-i-j-k-L*, in questo caso abbiamo che durante la propagazione del guasto il guasto scompare (valore *1* in posizione k) tale fenomeno è chiamato *scomparsa della D-frontiera*.
Infine, in Figura 12(d) vediamo la propagazione del guasto lungo il percorso *g-i-j-k-L* e vediamo come sia possibile sia portare il guasto fino all'uscita sia effettuare la *Line Justification* ottenendo così il vettore di test per il guasto analizzato. Tale vettore corrisponde al vettore *0-1-1-1* rispettivamente per gli ingressi *A-B-C-E*.
Abbiamo visto come questo meccanismo sia molto semplice tuttavia non è molto efficiente, in caso di circuito con *no_pi* numero di ingressi primari e *no_ff* numero di flip-flop del circuito e volendo simulare n porte logiche abbiamo che la la complessità è pari a :

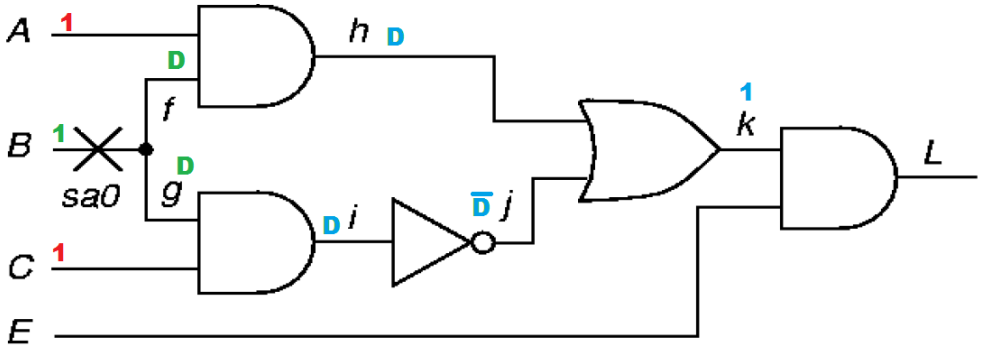
$$O(n \times 2^{no\_pi}\ times 4^{no\_ff})$$



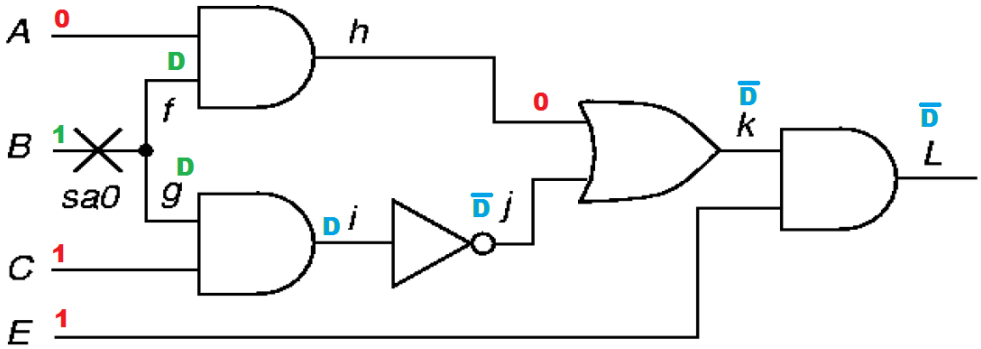
(a) Fault Sensitization



(b) Fault propagation sul percorso f-h-k-L



(c) Fault propagation parallela su f-h-k-L e g-i-j-k-L



(d) Fault propagation su g-i-j-k-L

Figura 12: Esempio di *Path Sensitization*

a \ b	0	1	X	D	$\overline{D}$
0	0	0	0	0	0
1	0	1	X	D	$\overline{D}$
X	0	X	X	X	X
D	0	D	X	D	0
$\overline{D}$	0	$\overline{D}$	X	0	$\overline{D}$

Figura 13: Forward implication table per una porta `AND`

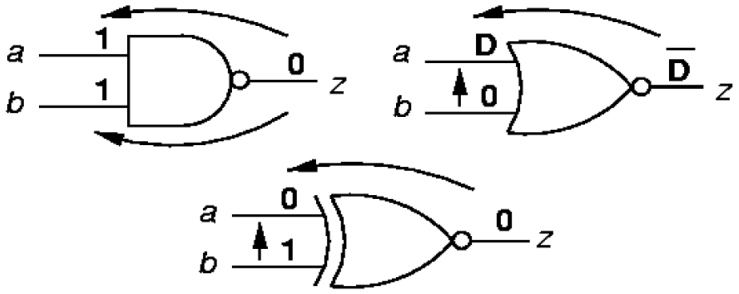


Figura 14: Esempi di backward implication

1.3.4 Algoritmo D

Prima di iniziare a parlare di questo algoritmo dobbiamo introdurre alcuni concetti. Il primo concetto che introduciamo è quello del *forward implication* la quale afferma che dati degli ingressi di porte logiche i quali sono etichettati in modo significativo è possibile determinare univocamente le uscite. Ad esempio per una porta `AND` la tabella della forward implication è mostrata in Figura 13.

Un'altra definizione è quella della *backward implication* molto simile alla forward solo che in questo caso i valori noti sono quelli dell'output e a volte di uno degli ingressi e si riesce a risalire alle uscite, un esempio è mostrato in Figura 14.

Durante la fase di decisione sui valori da assegnare agli ingressi può capitare che in alcuni punti possano essere possibili più scelte come ad esempio nel caso di una porta `AND` la cui uscita vale*0*. Per tenere traccia di queste possibilità e delle decisioni prese si utilizza un *implication stack* ovvero una tabella nella quale si tengono conto dei vari ingressi, del valore ad esso assegnati e se quel valore è l'unico possibile. Da questa tabella è possibile creare un *albero delle decisioni* per tenere traccia di quali percorsi sono esplorabili, quali sono stati esplorati e quali no. Un esempio di albero delle decisioni è mostrato in Figura 15 Questo tipo di algoritmo sfrutta il meccanismo del *Branch and Bound* per trovare una soluzione, ad ogni livello dell'albero l'algoritmo selezione uno dei possibili input escludendo al tempo stesso porzioni dell'albero per restringere le decisioni possibili in quanto un'esplorazione completa dell'albero è impraticabile.

L'algoritmo *D* è il primo algoritmo completo per la creazione di vettori di test, esso sfrutta oltre alle precedenti nozioni anche quella del *D-Cube* e del *D-Calculus*. Il *D-Cube* non è altro che il collassamento della tavola di verità del circuito utilizzando i valori dell'algebra di Roth.

L'algoritmo si compone di cinque passi

- Si numerano tutte le linee del circuito in modo incrementale dagli ingressi alle uscite.

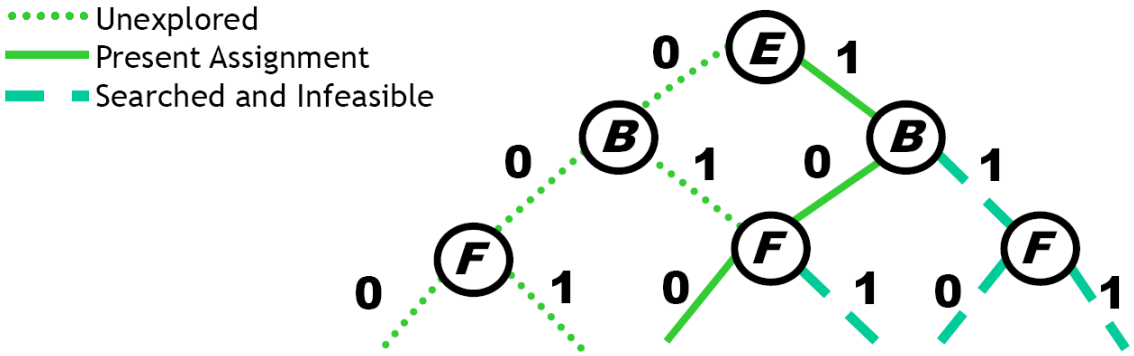


Figura 15: Esempio di albero delle decisioni

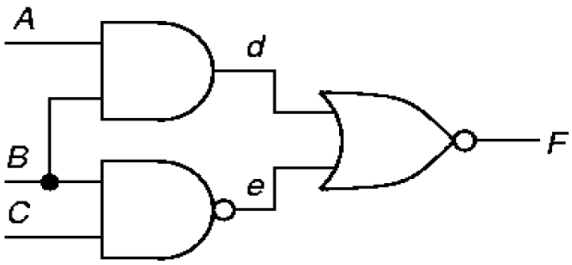


Figura 16: Circuito di esempio per l'algoritmo D

- Si seleziona una delle primitive 0che indicano il guasto dal *D-Cube*
- Si effettua il D-drive ovvero la propagazione del guasto
- Si verifica la consistenza
- Si ripete l'operazione del D-drive

Un esempio è quello in Figura 16 mentre la Tabella 2 è la tabella D-Cube di copertura Volendo analizzare uno *stuck-at 0* nel punto d i passi sono tre e sono mostrati in Tabella 3.

Uno *stuck-at 0* in d significa che in quel punto abbiamo una D utilizzando la backward implication otteniamo che sugli ingressi A e B abbiamo due 1. Sfruttando la seconda parte della Tabella 2 abbiamo che per un valore D sulla linea d abbiamo due possibilità (ultima e terzultima riga) prendiamo la prima delle due possibilità e questo ci dà il valore della linea e ovvero 0. A questo punto per avere 0 sulla linea e ed avendo già l'ingresso B fissato al valore 1 l'unica soluzione possibile è che all'ingresso di C abbiamo il valore 1. Per un ulteriore esempio più esaustivo si rimanda alle slide del corso e al video dell'insegnante.

A	B	C	d	e	F
1	1		1		
0			0		
	0		0		
	1	1		0	
	0			1	
		0		1	
				1	0
			1		0
			0	0	1
D	1		D		
1	D		D		
D	D		D		
	D	1		$\overline{D}$	
	1	D		$\overline{D}$	
	D	D		$\overline{D}$	
			D	0	$\overline{D}$
			0	D	$\overline{D}$
			D	D	$\overline{D}$

Tabella 2: Esempio di D-Cube di propagazione

Step	**A**	**B**	**C**	**d**	**e**	**F**	**Cube type**
1	1	1		D			Primitive D-cube of Failure
2				D	0	$\overline{D}$	Propagazione D-Cube
3		1	1		0		Backward implication

Tabella 3: Passi per la generazione del test

1.3.5 Algoritmo PODEM

L'algoritmo PODEM introdotto nel 1981 introdusse il concetto che l'albero delle decisioni non doveva essere espanso a tutte le linee del circuito ma solamente agli ingressi primari, inoltre esso non andava ad analizzare tramite backward e forward implication se il vettore esisteva ma semplicemente controlla di volta in volta l'esistenza della *D-Frontiera* ed infine utilizzava il *backtracing*.
L'algoritmo di PODEM si svolge in 6 passi:

1. Si assegna un valore binario ad uno degli ingressi ancora libero.
2. Si determinano le implicazioni su tutti gli altri ingressi
3. Il vettore di test è stato generato? Se si *fatto*.
4. Se è possibile assegnare valori ad altri ingressi tornare al punto 1
5. Esistono combinazioni di valori per gli ingressi non ancora testati? Se no il guasto è *non testabile.*
6. Cercare una combinazione di valori per gli ingressi tramite obiettivi e backtrace.

Per un esempio completo si rimanda alle slide del corso.

1.4 Generazione dei test nei circuiti sequenziali