Sistemi elettronici programmabili

Marco C

March 2025

Disclaimer

Questi appunti sono da prendere come riferimento per fare l'esame, ma per una conoscenza approfondita è meglio consultare il libro di Napoli da cui ho preso le informazioni, xoxo.

Chapter 1

Flusso di progetto per circuiti digitali

1.1 Front-end e back-end di un flusso di progetto

Il flusso di progetto per circuiti digitali programmabili ha una faccia del genere: La parte di front-end è orientata alla descrizione del circuito, implementazione

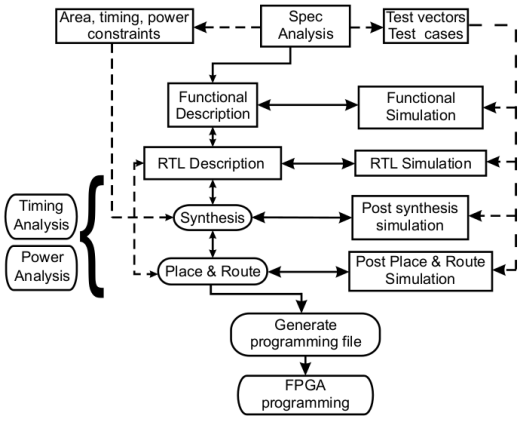


Figure 1.1: Flusso di progetto di un circuito digitale. Le fasi cerchiate in rettangolare le fa il progettista mentre quelle più rotonde sono semiautoatiche.

degli algoritmi e alla simulazione dello stesso. Essa si conclude con la definizione dei blocchi logici da utilizzare, che dipendono dall'offerta della tecnologia che si sta usando, unico caso in cui c'è dipendenza del front-end dalla tecnologia. Il back-end si occupa invece della progettazione e della verifica relativa alla strut-

tura del circuito, che è la progettazizione delle maschere per ASIC, lo stampo della PCB per i discreti eccetera.

1.2 Analisi delle specifiche

In questa fase si definiscono le specifiche del circuito da progettare. Per esempio in un device dedito al processing di immagini queste potrebbero essere la risoluzione, la profondità di bit etc...; successivamente si analizzano i costraint dati dal cliente quali possono essere area, tempi di propagazione e potenza dissipata. In particolare l'area dipende anche dalla tecnologia di FPGA scelta per il prototipo, perché si avranno limitazioni circa i blocchi digitali disponibili per organizzare il progetto. La terza cosa da fare è progettare in che modo verificare come si comporta il circuito in determinate situazioni e fare il test per la presenza di errori. Questa cosa viene fatta tramite l'introduzione dei test vector/test case, cioè input di cui si conosce l'output previsto, il quale si confronta con quello effettivo per verificare la presenza di errori.

1.3 Descrizione funzionale

A questo punto bisogna utilizzare un HDL (Hardware Description Language) per organizzare il progetto in uno schema a blocchi. Si tenga conto che non tutti i costrutti dell'HDL sono direttamente traducibili in circuito digitale. A questo punto si fanno uno o più *test bench* scritti in HDL. Un test bench corrisponde all'inieizione all'interno del circuito di un test vector e confrontare le uscite con i risultati attesi.

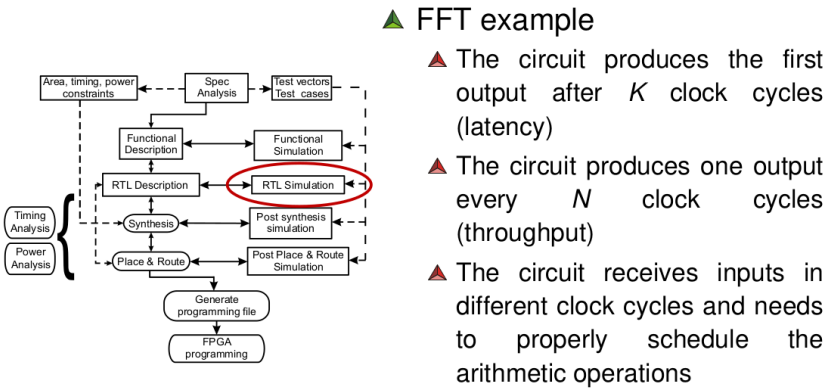
Esempio: il test bench per il circuito di elaborazione video applica al circuito, descritto per il momento in Verilog mediante costrutti non sintetizzabili, i segnali di test definiti in precedenza. Alcuni sono generati direttamente nel test bench, come ad esempio i video con immagini di colore uniforme, altri sono prelevati dalla memoria di massa e forniti al circuito da testare. Poiché si conoscono le uscite attese dal circuito nei casi considerati, il test bench controlla automaticamente la correttezza della simulazione ed evidenzia errori di progettazione. Eventuali errori, sulla suddivisione e sull'interazione dei moduli elementari, o sulle operazioni attribuite a ogni modulo, possono essere identificati e corretti in questa fase. Se la descrizione funzionale utilizza dei costrutti che non sono direttamente traducibili in un circuito, si avranno delle discrepanze tra questa simulazione e quelle successive. Di questo si dovrà tenere conto durante la progettazione del circuito.

Le uscite desiderate utilizzate per controllare il funzionamento del circuito possono: essere calcolate manualmente; fornite dal committente; generate utilizzando la descrizione software del circuito ottenuta dalla fase di analisi delle specifiche.

1.4 Descrizione RTL

In questa fase tutti i blocchi circuitali, anche quelli non sintetizzabili, devono essere espressi come circuito. Allora si sfrutta l'HDL con cui si è già fatto il

"design" ideale del progetto per ottimizzare le parti inutili e cercare di tradurre i blocchi logici non direttamente sintetizzabili. In questa parte l'HDL si occupa di tradurre i blocchi logici in costrutti formati da operatori, shift register eccetera. Inoltre, vengono definiti il numero di cicli di clock necesari per compiere determiante operazioni e assieme a quali sezioni del circuito è opportuno ripetere (tecnica di ridondanza). Per vedere se tutto è andato bene, si procede con la simulazione RTL per verificare che la transizione da HDL a circuito digitale è andato a buon fine. Spesso l'analisi funzionale e quella RTL sono fatte con lo stesso software in quanto è di solito compito del programmatore scegliere esplicitamente di effettuare quella funzionale, rendendo de facto simulazione RTL e funzionali sinonimi. La RTL è più veloce e non da informazioni sui parametri del circuito quali potenza o area occupata, ma fornisce dati circa il throughput, la latenza e la sincronizzazione tra i blocchi.



1.5 Sintesi

Nella fase di sintesi i blocchi funzionali del circuito digitale sono tradotti in componenti digitali. È divisa in due fasi. La prima, detta di *translate*, traduce i blocchi logici in componenti digitali totalmente generici che possono essere implementati da qualunque device FPGA che utilizza liberie standard per circuiti VLSI. La seconda fase, detta di *mapping*, associa ai componenti generici i componenti che sono effettivamente disponibili all'interno del dispositivo FPGA in esame. A volte c'è corrispondenza diretta, come un flip-flop per rappresentare un elemento ad un bit di memoria, mentre altre volte bisogna tradurre un blocco non sintetizzabile in altri modi, per esempio utilizzando una memoria per tradurre una logica combinatoria complessa. Alla fine della fiera si ottiene una *netlist*, ovvero una lista di connessioni sottoforma di file ASCII, che è un elenco degli ingressi, uscite e dei fili che collegano i vari componenti.

1.6 Simulazione post-sintesi

La simulazione post-sintesi viene fatta per:

- Individuare eventuali bug nella netlist, cioè errori nella fase di translate o di mapping
- Avere informazioni sul consumo di potenza, tenendo conto dei glitch (commutazioni multiple dello stesso segnale) e dei tempi di propagazione, che in questa fase possono misurati
- Se si è utilizzato il clock gating[1], questo si può simulare.

1.7 Place and route

Durante il place and route vengono piazzti (place) i componenti del circuito nelle zone a cui appartengono ed effettuati i collegamenti (route) fra di essi. Questa fase è dispendiosa in termini di potenza di calcolo perché, non esistendo algoritmi in grado di minimizzare l'area occupata e la lunghezza dei cavi conoscendo soltanto i pezzi da utilizzare, quello che si fa è try and error finché non si riduce man mano l'area. Questa cosa richiede uno sforzo di calcolo assurdo ed al progettista è chiesto soltanto, in questa fase praticamente automatica, di indicare quanto tempo e potenza di calcolo dare al programma per trovare il setup migliore, cioè quando questo deve sforzarsi. Tipicamente il massimo sforzo per il Place and Route si fa alla fine, quando si è sicuri di non dover modificare più niente. Come sempre si fa alla fine una simulazione post-P&R che segnala eventuali errori nelle connessioni. In questa simulazione qua l'accuratezza con cui si può calcolare potenza e tempi di propagazione è massima, perché gli effetti parassiti dei componenti si conoscono. Il check sul tempo di propagazione massimo, che si ha per il percorso critico, si fa durante la *time-analysis*, mentre quello per la potenza dissipata durante il *power-analysis*. Se il circuito non passa una di queste due, bisogna tornare indietro alla fase di progetto RTL e rivalutare le proprie scelte.

1.8 Generazione del file FPGA e programmazione

Se tutto è andato bene con la post-P&R analysis, si può generare il bitstream e caricarlo all'interno della FPGA, che viene programmato automagicamente.

[1] In computer architecture, clock gating is a popular power management technique used in many synchronous circuits for reducing dynamic power dissipation, by removing the clock signal when the circuit, or a subpart of it, is not in use or ignores clock signal.

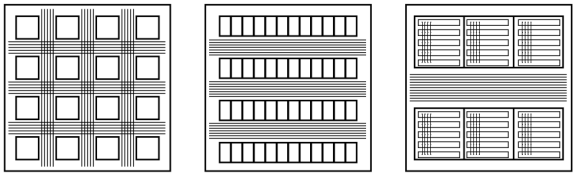
Chapter 2

Field Programmable Gate Array

Gli FPGA sono dispositivi elettronici programmabili, chiamati così perché derivanti dai Gate Array e programmabili direttamente sul proprio tavolo da lavoro. L'utlizzo della parola "programmare" riferita ad un FPGA è fuorviante, in quanto un progettista FPGA si occupa di organizzare i collegamenti del circuito e la scelta dei componenti, ottenendo come risultato un circuito digitale, a differenza di qualcuno che programma in linguaggi come il C o l'assembly, per i quali "programmare" significa elencare una serie di istruzioni da fare eseguire ad un computer.

2.1 Struttura di un FPGA

Ci sono tre layout principali per la struttura di un FPGA: La disposizione a ma-



(a) Array simmetrico. (b) Row based. (c) Gerarchico.

Figura 4.5: Possibili organizzazioni della logica e delle risorse di routing in un dispositivo FPGA.

trice e quella a righe sono alquanti simili e differiscono solo per il fatto che nella prima i percorsi logici si intrecciano ad ogni intersezione, mentre per la seconda sono compresi fra due righe di elementi logici. La terza, quella gerarchica, è fondamentalmente diversa. La struttura è a righe, ma ogni blocco logico ha una sottostruttura a righe di media complessità logica costituita da blocchi logici a loro volta costituiti da una sottostruttura e così via.

Le FPGA si possono distinguere anche a seconda del fatto che siano programmabili una sola o più volte, oppure che siano volatili e non:

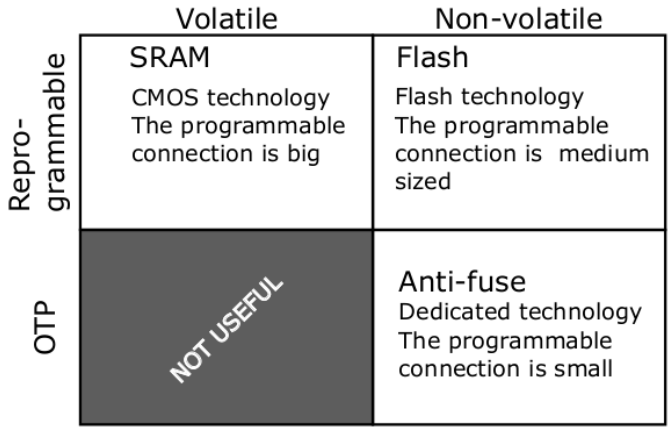
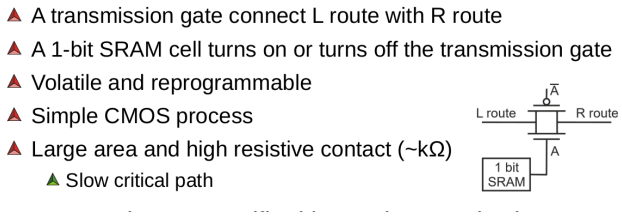


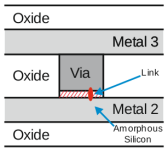
Figure 2.1: I vari modi per distinguere un FPGA

Un esempio di FPGA programmabile più volte e volatile è quello basato su SRAM. Il bitstream viene caricato e ogni celletta di SRAM conserva un dato che identifica se una connessione deve essere aperta o chiusa tramite porta di trasmissione CMOS.



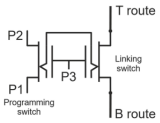
È necessaria una memoria flash esterna per conservare la configurazione, che viene caricata in memoria dinamica ogni volta.
Un esempio invece di non-volatile OTP è la tecnologia FPGA basata su anti-fuse. In questo caso si crea un piccolo vuoto nel canale di via tra due layer di metal e dove si vuole fare il collegamento si applica una ddp abbastanza forte che scioglie la via e collega i due layer di metal.

- Antifuse connection
  - Two metal regions with thin dielectric between them
  - Programming voltage applied between metal regions
  - The current flowing into metals and via creates the link
  - Non-Volatile and OTP: once programmed, connection preserves over time and never changes
  - CMOS process + further steps
  - Contact has minimum area
  - Low resistive contact (~80Ω-500Ω)



Un esempio invece di FPGA non-volatile e riprogrammabile è dato da quelli basati su flash switch:

- Flash Switch connection
  - Usage of flash mosfets with shorted floating gates
  - Non-volatile and reprogrammable
  - Complex production process
    - High costs
  - Lower area than SRAM-based connection
  - External flash memory is not required
  - Resistive behavior comparable to SRAM solution



2.2 Architettura di celle logiche programmabili

Al fine di garantire la massima flessibilità, gli elementi logici che costituiscono un FPGA sono anch'essi programmabili. Ci sono due tipi di architetture per le celle logiche: quelle basate su multiplexer e quelle basate su Look Up Tables (LUT).
Le FPGA che utilizziamo noi sono basate su Look-up table. Una LUT è un blocco logico ad n ingressi ed una uscita. La LUT ha un elemento di memoria per conservare 2^n bit che sono i possibili che può assumere una funzione booleana ad n ingressi, potendo implementare potenzialmente 2^{2^n} funzioni diverse.

Un moderno FPGA si compone dei seguenti elementi:

- I **Logic Element** (LE), implementati come abbiamo detto in MUX o LUT. Possono essere piccoli blocchi RAM, ROM, shift register...
- I **blocchi RAM** (BRAM), slice di unità o decine di kilobyte di RAM
- I **blocchi DSP**, cioè quelli per il digital signal processing come accumulatori, addizionatori, moltiplicatori...

- CPU, che può essere *soft*, cioè implementata tramite i blocchi logici del dispositivo, oppure *hard*, fatta ad-hoc per il device. Le prime sono più lente ma possono essere customizzate a seconda delle necessità.

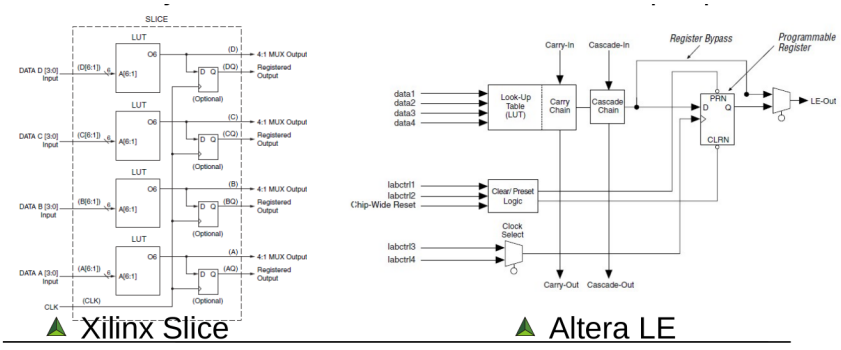
Sulla parte periferica del chip FPGA troviamo:

- I **banchi I/O** che sono altamente programmabili per tutte le stronzate che ti passano in mente: gestione della reiezione del rumore, direzione dell'IO, valori di corrente eccetera.
- Il **clock management**, che si occupa di trasferire il segnale di clock all'FPGA. Il CM può ritardare, moltiplicare e fare altre operazioni sul segnale di clock.
- **High speed IO** per il trasferimento seriale ad alta velocità.

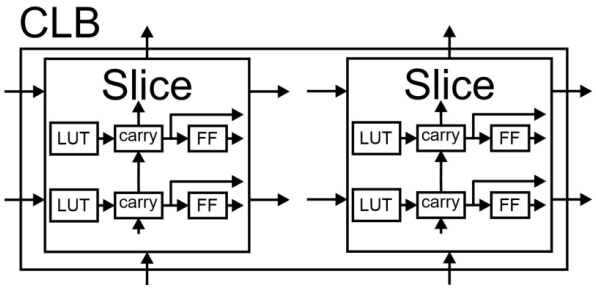
2.3 Caratteristiche delle celle logiche programmabili

Gli elementi sequenziali sono d'importanza fondamentale. Sebbene si possa pensare in un primo momento di poter implementare un flip flop tramite invertitori classici e collegamenti programmabile, ciò riduce il numero di celle logiche disponibili e da molta difficoltà nella gestione del timing. Allora i programmatori di FPGA preferiscono implementare i Flip-Flop nativamente nei loro logic element (LE). Questi flip-flop infatti devono essere pianamente programmabili e si deve poter decidere su quali fronti di clock farli lavorare, se usare un segnale S/R eccetera.

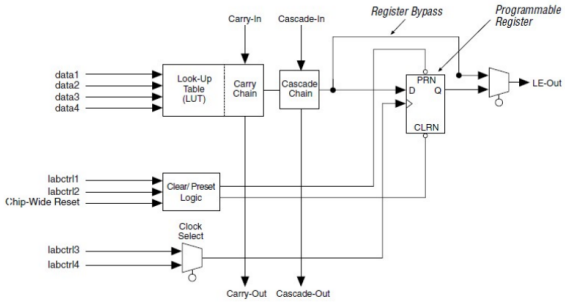
Generalmente gli LE implementano funzioni combiantorie. In alcune schede gli LE sono chiamati Slice. Dunque ogni Slice/LE ha delle LUT e dei flip-flop:



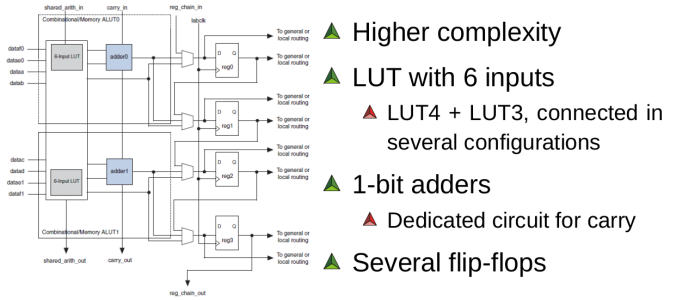
- Xilinx solution
  - Each CLB comprises several Slices
  - Each Slice comprises at least a LUT and a flip-flop
  - Slice can be used stand-alone or cascaded



- Altera solution
  - Each LE comprises a LUT and a flip-flop
  - Several LEs are collected in a Logic Array Block (LAB)



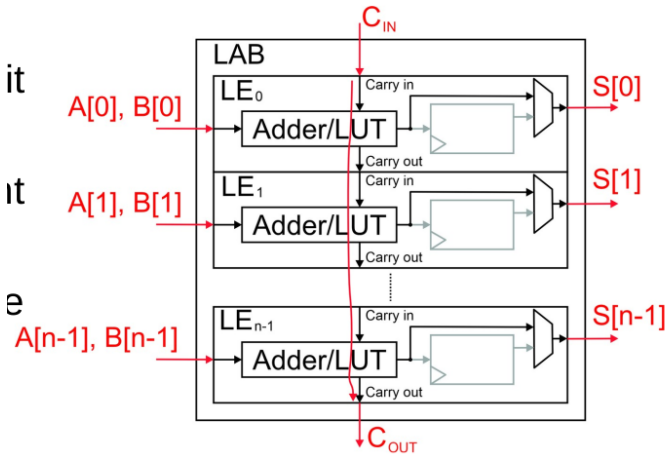
Le FPGA ad alte prestazioni implementano gli ALM (Adaptive Logic Module):



- Higher complexity
- LUT with 6 inputs
  - LUT4 + LUT3, connected in several configurations
- 1-bit adders
  - Dedicated circuit for carry
- Several flip-flops

Questo tipo di elemnti logici hanno al loro interno LUT adattabili, nel caso di Altera si può scegliere di utilizzare LUT4, LUT5 o LUT6 collegando in cascata delle LUT4.

Gli addizionatori nei circuiti digitali sono comuni e si può implementare un adder da un bit utilizzando una LUT4 divisa in due LUT3, una per il risultato della somma di due bit e l'altra per il riporto. Mettendone in cascata n, si ottiene un addizionatore ad n bit. Negli ALM ci sono circuiti addizionatori dedicati,

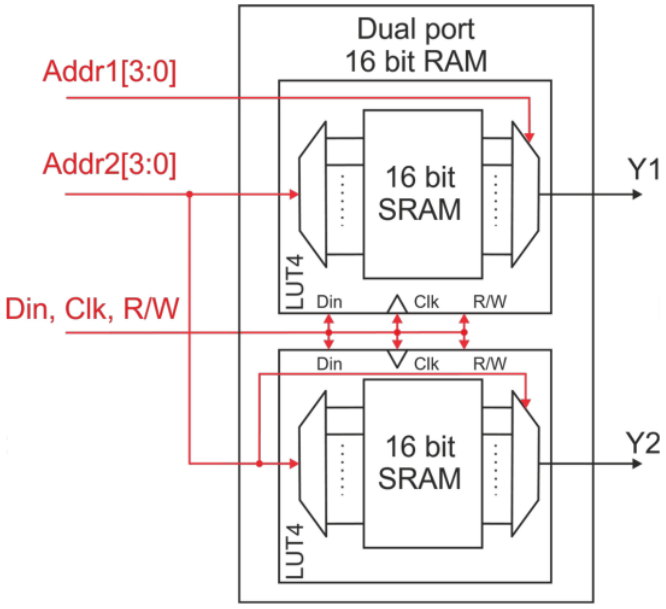


per garantire migliore flessibilità. Ci sono anche circuiti per la propagazione del riporto (carry-propagation), dunque connettendo n ALM in cascata è possibile creare addizionatori di $2n$ bit.

Le look up table si comportano come piccole memorie RAM e possono essere programmate tramite i registri ProgDIN, ProgCLK, ProgWR, i quali una volta programmata la cella vengono disabilitati, facendo sì che la LUT si comporti come un circuito combinatorio ordinario. Si possono usare allora le LUT come memorie RAM distribuite, i cui bit possono essere scritti a runtime tramite i registri citati sopra.

- **ProgDin**: registro dato da memorizzare nella RAM.
- **ProgCLK**: registro per temporizzare operazioni di lettura e scrittura.
- **ProgRW**: registro per segnalare stato di lettura o scrittura.

Le LUT possono essere utilizzate anche per fare RAM dual port e performare lettura e scrittura in contemporanea:



Per la costruzione di registri a scorrimento, tipicamente si mettono in serie dei flip-flop. Altera ottimizza questa cosa con una circuiteria ad-hoc per le cascade di ALM, mentre altre schede permettono di creare shift-register programmando le varie RAM basate su LUT per tale scopo. In generale con una LUT da n bit si può implementare un registro a scorrimento di 2^n locazioni di memoria.

Chapter 3

Circuiti combinatori

3.1 Multiplexer

Un multiplexer (abbreviato in MUX) è un circuito combinatorio. Un circuito combinatorio è un circuito la cui uscita dipende esclusivamente dagli ingressi e non dallo stato corrente del circuito, come invece accade nei circuiti sequenziali. Per questa loro proprietà, i circuiti combinatori non implementano alcun tipo di retroazione.

Un MUX, nella sua forma più generale possibile, è un circuito che ha in ingresso n stringhe da p bit. Il mux instrada uno tra gli n ingressi tramite p bit di selezione e viene detto *completo* se $2^m = n$. Per implementare un mux di stringhe a p bit basta mettere in parallelo p mux $n : 1$, ognuno controllato dagli stessi m bit e con le uscite preelvate in parallelo:

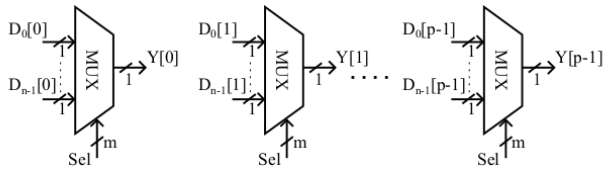


Figura 6.2: Il Mux con ingressi su più bit si realizza componendo Mux con un solo bit per ingresso.

Il MUX $2 : 1$, il più utilizzato nei circuiti digitali, equivale al costrutto "if-else". Un esempio lo abbiamo con il calcolo del valore assoluto. Se A è il valore del numero ed S il segno di A, il MUX $2 : 1$ implementa una funzione del genere (avendo dato gli ingressi opportuni):

$$Y = A\bar{S} + \bar{A}S \tag{3.1}$$

cosicché se A è positivo (ed il bit di segno è zero, cioè "+") allora ritorna A, altrimenti ritorna il valore di $-A = \bar{A}$, che è positivo se A è negativo, così come il bit di segno S che sarà uguale ad 1.

3.2 Decoder binario

Il decoder binario è un altro circuito altrettanto utilizzato e permette di convertire l'espressione di numeri da un formato a quello binario: Un utilizzo pratico

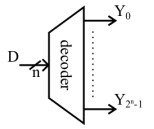


Figura 6.4: Il simbolo del decodificatore binario.

D (in decimal)	Y_0	Y_1	Y_2	$\cdots$	Y_{2^n-2}	Y_{2^n-1}
0	1	0	0	$\cdots$	0	0
1	0	1	0	$\cdots$	0	0
2	0	0	1	$\cdots$	0	0
$\vdots$	0	0	0	$\ddots$	0	0
2^n-2	0	0	0	$\cdots$	1	0
2^n-1	0	0	0	$\cdots$	0	1

Tabella 6.2: Tabella di verità del decodificatore binario.

di un decoder binario lo si trova nelle memorie, dove vengono utilizzati per selezionare le colonne e le righe della matrice di locazioni, che corrisponde all'attivazione dei MOS di riga e di colonna.

3.3 Encoder

Un codificatore (encoder) è un qualsiasi circuito che ha in ingresso un codice e lo traduce in un altra rappresentazione. La lunghezza della stringa d'uscita è tipicamente minore di quella d'ingresso, ragion per la quale un encoder non può associare ad ogni ingresso un valore unico di uscita. Per tale ragione, si deve agire in uno dei seguenti modi: o si garantisce che in ingresso al circuito non ci siano ingressi non ammessi oppure si fa in modo che tali ingressi, quando presenti, siano scartati, aggiungendo dei bit all'uscita che ne certifichino l'ammissibilità.
Un esempio di encoder è quello *one-hot*, che ritorna (in binario) la posizione dell'unico uno all'interno della stringa:

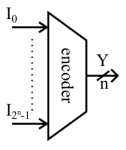


Figura 6.5: Il simbolo del codificatore binario.

I_0	I_1	I_2	$\cdots$	I_{2^n-2}	I_{2^n-1}	Y (in decimal)
1	0	0	$\cdots$	0	0	0
0	1	0	$\cdots$	0	0	1
0	0	1	$\cdots$	0	0	2
0	0	0	$\ddots$	0	0	$\vdots$
0	0	0	$\cdots$	1	0	2^n-2
0	0	0	$\cdots$	0	1	2^n-1
all other input combinations						don't care

Figura 6.6: Tabella di verità del codificatore.

L'encoder può essere anche a priorità e viene chiamato in quel caso *priority encoder*. Un priority encoder risolve il problema degli ingressi non validi con più di un bit alto cambiando l'uscita associata alla configurazione con un solo

bit alto che ha la priorità maggiore. Se per esempio entra 110 ed 100 ha la priorità su 010, allora l'uscita sarà quella di 100. Questo meccanismo trova utilizzo nella gestione delle interrupt dei processori, dove il concetto di priorità è di fondamentale importanza. L'enoder a priorità ha lo stesso simbolo dell'encoder normale, eccetto per il segnale di "Idle" disegnato sopra. Il segnale di "Idle" serve a notificare quand'è che l'encoder ha tutti zero in ingresso.

3.4 Comparatore

Un comparatore è un circuito che fa un confronto fra due numeri. I comparatori che implementano confronti come l'uguaglianza fra due numeri sono semplici da realizzare, mentre la complessità cresce più che linearmente con l'aumentare dei bit se si prendono comparatori che verificano che un numero sia maggiore di un altro. I comparatori sono di fondamentale importanza nell'implementazione dei cicli, del confronto fra chiavi di sicurezza nella crittografia eccetera.

Un esempio è il comparatore che verifica che un numero sia maggiore di zero. La funzione booleana che implementa questa cosa è:

$$IsZero = \bar{I_0}\bar{I_1}\cdots\bar{I_{n-1}} = \overline{I_0 + I_1 + \cdots + I_{n-1}} \tag{3.2}$$

In VLSI la forma NOR è preferita (cioè come prodotto di negati) perché utilizza $2n$ transistor per una funzione ad n ingressi, contro i $4n+2$ utilizzati dalla logica NAND.

Se invece si vuole effettuare il confronto con una costante C diversa da zero, bisogna tener conto di caso per caso. Se la costante, una stringa di n bit, ha meno di $(n+1)/2$ alti, conviene ancora utilizzare la logica NOR, altrimenti è conveniente operare in NAND negando le uscite necessarie. Supponiamo per esempio che $C = 110$, allora la funzione da implementare è:

$$Y = I_2 \cdot I_1 \cdot \bar{I_0} = \overline{\bar{I_2} + \bar{I_2} + I_0} \tag{3.3}$$

In questo caso conviene una NAND per esempio, mentre se fosse stato, ad esempio, $C = 10001$ sarebbe convenuta una NOR.

Per verificare che due segnali A e B siano uguali, bisogna fare il prodotto delle XNOR fra i bit di A e B:

$$Y = \Pi_i^n (A_{i-1} \odot B_{i-1}) \tag{3.4}$$

dove ricordiamo la XNOR è il negato della NOR, dunque è alta se i due bit sono uguali.
Per relazioni maggiore/minore, ad esempio $A > B$, il circuito è più complesso e richiede oltre s XNOR e AND svariate porte OR.

3.5 Full adder

Un full-adder, cioè un addizionatore completo, è un circuito che ha in ingresso tre bit, A, B, C_{in}, con C_{in} il riporto in ingresso, e da in uscita su due bit la somma S e il riporto di uscita C_{out}. Ovviamente il bit somma ha più peso del carry di uscita e dunque il full-adder viene spesso chiamato compressore $3:2$, perché comprime un informazione su 3 bit di peso unitario in una su due bit di peso diverso.

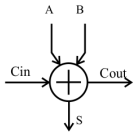


Figura 6.9: Simbolo e port di ingresso e uscita del full adder.

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figura 6.10: Tabella di verità del full adder.

Chapter 4

Introduzione al Verilog

4.1 Caratteristiche di un HDL

Il Verilog è un HDL e dunque permette la descrizione di circuiti digitali e di comportamento, a seconda del livello di astrazione con cui si lavora. Utilizzare un HDL conviene proprio perché si possono descrivere funzioni booleane nel modo più low level possibile, così come sono molto utili i costrutti per la descrizione comportamentale (behavioral constructs). Assieme al VHDL, il Verilog è uno standard industriale per la progettazione di FPGA, CPLD ed ASIC.

Cosa rende un HDL diverso (e più adatto per il low level) da un linguaggio di programmazione software è la possibilità di eseguire istruzioni in parallelo, cosa fondamentale nei circuiti e non presente nei linguaggi software che sono rigorosamente sequenziali. Gli HDL possono essere utilizzati per simulare il comportamento del circuito includendo anche i ritardi, tramite costrutti di simulazione, inoltre sono capaci di effettuare la sintesi circuitali tramite appositi costrutti di sintesi.

L'HDL può essere utilizzato anche per il calcolo di vettori di test, ma quando si passa alla descrizione RTL bisogna ripulire il codice Verilog da costrutti non sintetizzabili. Nelle fasi di sintesi, P&R e avanti non viene più scritto codice HDL.

4.2 Come si descrive un circuito

Tutti i file verilog che descrivono circuiti iniziano con la keyword *module*.

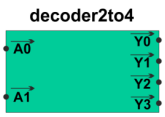
Each module contains:

module module_name (port_list);
declarations: reg, wire, parameter, input, output, inout, function, tasks,...
Statements: initial, always, module instantiation, continuos assignment...
endmodule

Figure 4.1: Initial è un esempio di costrutto non sintetizzabile che va rimosso in fase di sintesi

Un esempio facile è il decoder 2:4 Prima di scrivere il codice va sempre fatto lo

Verilog example




```
module decoder2to4 (A1,A0,Y3,Y2,Y1,Y0);
input  A1,A0;
output Y3,Y2,Y1,Y0;
// module, input, and output declarations derive from the top level
wire A1,A0,Y3,Y2,Y1,Y0;

assign Y0 = (~A1)&(~A0); // The Boolean functions
assign Y1 = (~A1)&( A0);
assign Y2 = ( A1)&(~A0);
assign Y3 = ( A1)&( A0);

endmodule
```

- Statements order is not important for the circuit
- Concurrent statements

schema a blocchi. Partiamo per esempio dalla tabella di verità e ricaviamo:

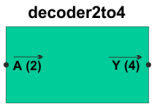
$$y_0 = \bar{a_0}\bar{a_1} \tag{4.1}$$
$$y_1 = a_0\bar{a_1} \tag{4.2}$$
$$y_2 = \bar{a_0}a_1 \tag{4.3}$$
$$y_3 = a_0a_1 \tag{4.4}$$
$$\tag{4.5}$$

e quindi la implementiamo all'interno del codice. Allora mi servono 4 LUT per fare le AND da due e 2 LUT per fare gli inverter.

Lo stesso circuito si può descrivere con i bus array, definendo quindi due vettori A ed Y che sono i bus rispettivamente di entrata ed uscita:

Signal arrays

```
module decoder2to4 (A,Y); // Direct derivation of the top level schematic
input  [1:0] A; // These two lines also come from the top level
output [3:0] Y;
wire   [1:0] A;
wire   [3:0] Y;
```




```
assign Y[0] = (~A[1])&(~A[0]); // The Boolean functions
assign Y[1] = (~A[1])&( A[0]);
assign Y[2] = ( A[1])&(~A[0]);
assign Y[3] = ( A[1])&( A[0]);

endmodule
```

Syntax : [<start>:<end>]

```
wire [3:0] sel;      // 4 bit bus
input [7:0] add;     // 8 bit bus
input [1:3] add;     // 3 bit bus
```

4.3 Descrizione gerarchica

La descrizione gerarchica è conveniente perché è complicato lavorare con codici lunghi e anche perché i pezzi che compongono la struttura totale possono essere riutilizzati[1].
La keyword che consente di richiamare all'interno di un top level (circuito a gerarchia maggiore) un circuito lower level (un sottocircuito più elementare) é:

$$< modulename >< istancename > (< interface\ ports >)$$

Il modulo da mette è quello lower level, poi va istanziato un oggetto che ha le caratteristiche di quel modulo e infine vanno messe le interface ports, che descrivono in che modo si collegano i segnali IO del modulo istanziato.

[1] equivalente della modularità per programmi software

4.4 Blocchi procedurali

I costrutti verilog sono concorrenti: l'esecuzione dei costrutti è parallela e l'ordine con cui vengono dichiarati i costrutti non variano il funzionamento. Le istruzioni all'interno del blocco procedurale, però, sono sequenziali. Post-sintesi non c'è più il blocco procedurale ma un circuito.

Un segnale prodotto da un blocco procedurale deve essere definito come *reg*,

Procedural blocks

- Syntax

```
always @(<event control>)
begin
        <procedural statements>
end
```

- *always* : indicates the begin of a procedural block.
- The instructions embraced by the *begin end* keywords are executed sequentially
- @ is followed by a logic condition ('event control' also named as 'sensitivity list') that, when true, enables the execution of the procedural block.

ma ciò non vuol dire che *reg* sia un registro, semplicemente è un segnale che esce da un blocco procedurale.

- Other important characteristics:
- Every signal whose value is set inside a procedural block must be declared as *reg* type.
- When a circuit is being described the assignment of a value to a signal uses the following syntax:
  A <= B;
- The assignment with the <= symbol is indicated as *non blocking*.

Procedural blocks: sensitivity list

- In some cases, the sensitivity list characterizes the entire circuit
  - flip-flop : the clock edge determines the change of the Q (output) signal
- Functional simulation
  - The sensitivity list determines if the procedural block must be executed. When the sensitivity list is missing the procedural block runs continuously and slows down the simulation
- A digital circuit depends on the changes of the input signals
  - The sensitivity list indicates to the synthesizer the events that activate the circuit and help defining the circuit without any ambiguity.

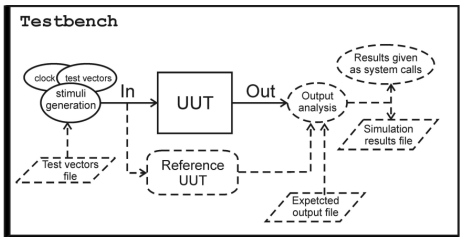
Un esempio dove è corretto utilizzare una sensitivity list incompleta è quello del flip flop di tipo D sul fronte di salita. I segnali d'ingresso sono il dato ed il clock, ma l'ingresso può variare soltanto sul fronte di salita del clock (cosa stabilità all'interno del verilog con la keyword *posedge*):

▲ Typical example of circuit whose unambiguous description is the key to let the synthesizers use the high performance flip flops yet present into the FPGA (or in the ASIC standard cell library).

```
module df (d,q,clk); // Module definition
input wire d,clk;
output reg q;   // Assigned in a procedural block -> reg type

always @(posedge clk) // Procedural block. Activated by the
                      // positive edge of the clock
begin
q <=d; // D flip flop
end
endmodule
```

4.5 Test bench

Il testbench non è un circuito ma un programma di test, quindi non ha né ingressi né uscite, quindi va dichiarato il module senza alcun argomento. Tutto quello che fa il testbench è pipare degli stimoli e verificare le uscite del circuito.



Esempio testbench di un priority encoder 8:3 L'istruzione timescale da le unità

```
`timescale 1ns/1ps    // define simulation time units
module prenc83_tb();  // module with no inputs or outputs
reg  [7:0] Atb;  // reg net type. Assigned in initial block
wire [2:0] Ytb;
wire Idletb;

prenc83 UUT(.A(Atb),.Y(Ytb),.Idle(Idletb)); // UUT

initial        // Just once at the beginning of the simulation
begin
Atb = 8'h01;   // Eight bits hexadecimal value
#100;          // 100 time units delay -> 100 ns
Atb = 8'h02; #100;
Atb = 8'hF0; #100;     // repeat input assignment at will
$stop;         // System call that stops the simulation
end
endmodule
```

di tempo di riferimento. ALl'interno dell'initial, #100 si riferisce ad "aspetta 100 nanosecondi" perché il timescale ha il nanosecondo come unità di misura. La seconda unità di misura, il picosecondo, indica l'ultima cifra significativa da considerare per i valori del testbench. La sesta istruzione serve a collegare

i segnali IO del circuito: i segnali collegati agli ingressi del testbench vanno definiti come reg, mentre quelli alle uscite come wire. Questa cosa è intuitiva perché i segnali d'ingresso devono stimolare i blocchi procedurali.

Chapter 5

Circuiti aritmetici

5.1 Richiami sulla rappresentazione dei numeri

Un numero a virgola fissa senza segno si indica con la notazione:

$$U_{n,m} \tag{5.1}$$

dove n è la potenza di 2 assegnata all'MSB della parte intera ed m la potenza *negativa* di 2 assegnata all'LSB della parte frazionaria. Per esempio $U_{4,3}$ ha MSB di peso 2^4 ed LSB di peso 2^{-3}, dunque $U_{4,3} \in [0, 31.875]$.
Per rappresentare invece numeri negativi a virgola fissa c'è il complemento a due. Un numero in complemento a due è indicato con:

$$Q_{n,m} \tag{5.2}$$

dove Q ricorda l'insieme dei numeri razionali. I parametri n ed m sono rispettivamente la potenza dell'MSB (numero negativo) e la potenza *negativa* dell'LSB della parte frazionaria

$$Q_{n,m} = -2^{-n}A_{n+m}\ 2^{n-1}A_{n+m-1}\ \cdots 2^{-(m-1)}A_1\ 2^{-m}A_0 \tag{5.3}$$

Per esempio

$$Q_{2,0} = 1010 = -2 \tag{5.4}$$

5.2 Rappresentazione in Verilog

Nel verilog, un wire può assumere il significato di un segnale o di un numero a seconda del contesto. Se per esempio A è una stringa di 8 bit che entra in un encoder, allora si starà trattando di un segnale, per esempio un vettore di priorità. Viceversa, se A entra in un sommatore dobbiamo aspettarci che abbia valore aritmetico e non logico.

- *wire* and *reg* are considered as unsigned numbers
- In the 2001 Verilog standard the signed *wire* and *reg* have been introduced.
- `wire signed [7:0] A;`
- `reg signed [15:0] B;`
- Operations including unsigned numbers give unsigned result.
- Operations including only signed numbers give signed result.
- $signed() and $unsigned() system functions convert type number
- The `signed` keyword can be omitted in most cases.

5.3 Addizionatore binario

Un addizionatore binario è un circuito che ritorna la somma aritmetica degli ingressi. I bit di ingresso sono tre, i due operandi A e B ed il bit di riporto in entrata C_{in}. I bit di uscita sono la somma S, il segnale di overflow OFL ed il riporto in uscita C_{out}. Il riporto in uscita può essere usato come l'$n+1$-esimo bit della somma ma non può essere usato per segnalare l'overflow (si può avere OFL con ambo i valori di Carry in uscita).
Si possono ottenere addizionatori a $2n$ bit mettendone in cascata 2 da n bit e collegando il C_{out} del primo al C_{in} del secondo. I riporti in entrata degli addizionatori al primo stadio sono posti a massa.

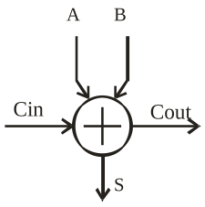
- From the theory of 2's complement binary numbers we know that the most significant bits (A_{n-1} and B_{n-1}) also represent the sign bit.
- There is an overflow in two cases
  - Sum 2 positive numbers (>0) and the result is <0: $A_{n-1}=B_{n-1}$='0' and S_{n-1}='1'
  - Sum 2 negative numbers (<0) and the result is >0: $A_{n-1}=B_{n-1}$='1' and S_{n-1}='0'
- The Boolean equation for the overflow bit is then:

$$OFL = A_{n-1} \cdot B_{n-1} \cdot \bar{S}_{n-1} + \bar{A}_{n-1} \cdot \bar{B}_{n-1} \cdot S_{n-1}$$

- It requires that the sum bits are known. It tends to slow down the adder
  - adds a logical gate to the critical path for the calculation of S_{n-1}.

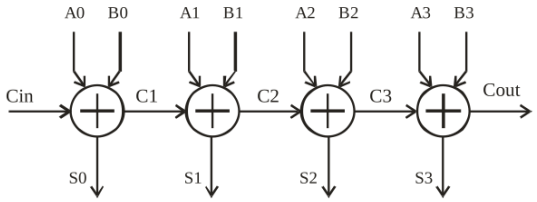
- 1 bit adder – also known as 3-2 compressor

Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Se ci sono n addizionatori in cascata, il tempo totale di delay tra l'ingresso del primo stadio e l'uscita dell'ultimo è pari a: Un addiziontore del tipo di sopra

▲ Delay : $T_{CR} \sim n * T_{Cin\text{-}Cout}$



è detto **carry ripple**, cioè a propagazione di riporto. Esso consta in n addizionatori ad un bit, con l'uscita che viene presa in parallelo da ogni adder, collegato al precedente tramite il carry out. Questo è il circuito più semplice da realizzare per costruire un addizionatore, ha il minor consumo di potenza dissipata ma è anche il più lento con tempo pari a T_{CR}.

▲ An equivalent equation to check the overflow for a n bit adder is:
OFL=Cin(n-1) XOR Cout(n-1)

▲ checking that the Cin and Cout of the most significant full adder agree.

$C_{in}(n-1)$	A(n−1)	B(n−1)	S(n−1)	$C_{out}(n-1)$	OVFL	$C_{in}(n-1) \oplus C_{out}(n-1)$
0	0	0	0	0	NO	0
0	0	1	1	0	NO	0
0	1	0	1	0	NO	0
0	**1**	**1**	**0**	**1**	**YES**	**1**
1	**0**	**0**	**1**	**0**	**YES**	**1**
1	0	1	0	1	NO	0
1	1	0	0	1	NO	0
1	1	1	1	1	NO	0

Per la sottrazione, il circuito è lo stesso del sommatore binario. La differenza $A - B$ è scritta come $A + (-B)$ dove $-B$ si ottiene negando B bit per bit ed aggiungendo 1, per esempio:

$$1110_2 = (-2)_{10} \implies 0001_2 + 0001_2 = 2 \tag{5.5}$$

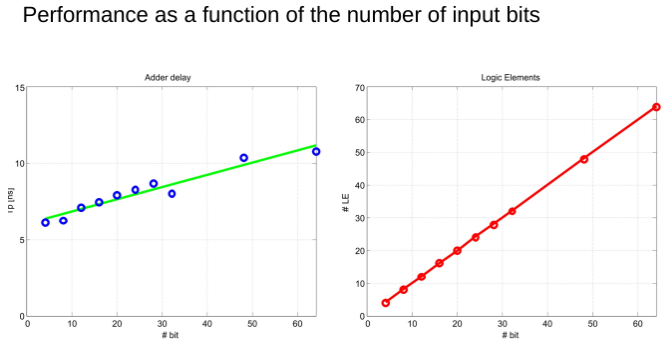
5.4 Topologie di addizionatori per FPGA

La topologia utilizzata per l'adder binario è quella a **carry ripple** con un piccolo fan-in. I blocchi logici delle moderne FPGA includono logica orientata alla propagazione del riporto, atta ad ottimizzare i collegamenti circuitali dei sommatori a carry ripple. Gli adder a carry ripple occupano molto poco spazio, dunque è possibile trovarne tantissimi anche nei più piccoli FPGA.

Negli adder carry ripple i bit di carry out dello stadio $n-1$ e di quello n possono essere utilizzati per il check dell'overflow: se entrambi sono alti, allora c'è sicuro overflow:

$$OFL = C_{out,n-1} \oplus C_{out,n} \tag{5.6}$$

Le prestazioni del circuito addizionatore si misurano in base al numero di Logic Element impiegati ed il ritardo di CR, entrambi funzione del numero di bit dell'addizionatore:



5.5 Operazioni comuni

Molto spesso gli ingressi dei circuiti digitali sono numeri rappresentati in virgola fissa ma con un differente numero di bit o con la virgola in posizioni differenti. Le operazioni aritmetiche più comuni sono quelle che trasformano la rappresentazione di un numero eliminando o aggiungendo dei bit a destra o a sinistra della rappresentazione originaria. L'operazione sembra banale ma serve ad allineare la virgola di rappresentazioni diverse, a evitare e controllare i casi di overflow, a determinare la precisione di un'operazione aritmetica.

10.6.1 Aggiungere bit a destra del numero

Quando si vogliono sommare due numeri con risoluzione diversa e non si vuole perdere risoluzione sul risultato, è necessario estendere la rappresentazione del segnale con la risoluzione più bassa aggiungendo dei bit. L'operazione equivale ad allineare la posizione della virgola con quella del numero a maggiore risoluzione e si implementa aggiungendo degli zeri a destra del numero. Ad esempio, il numero 5'b11011 in rappresentazione U(2,2) si porta nel formato U(2,4) aggiungendo due zeri in coda: 7'b1101100. Si noti come la posizione della virgola nella rappresentazione non sia cambiata e il numero sia inalterato. In Verilog l'operazione si descrive mediante l'operatore di concatenazione.

```
wire [4:0] A;    // U(2,2)
wire [6:0] B;    // U(2,4)
assign B={A,2'b00};
```

L'operazione non richiede l'implementazione di un circuito ma solo il cambio di nome di alcuni fili e l'aggiunta di fili collegati a massa come nell'esempio di figura 10.15.

Figura 10.15: Estensione di un numero con due zeri meno significativi.

Figura 10.16: Troncamento dei tre bit meno significativi di un segnale a 7 bit.

Talvolta si desidera ridurre la precisione di un numero per evitare che il numero di bit del risultato di un calcolo sia troppo grande o per rispettare i vincoli su un interfacciamento, e quindi bisogna eliminare i bit meno significativi del numero effettuando un'operazione di troncamento o di arrotondamento. Ad esempio, si utilizza il troncamento o l'arrotondamento se il risultato di un'operazione è su 16 bit mentre il registro nel quale bisogna memorizzare il risultato è da 8 bit.
Il troncamento è il modo più semplice per eseguire l'operazione e consiste nello eliminare i bit che non si utilizzano. Un minimo di riflessione mostra che l'errore dovuto al troncamento è sempre dello stesso segno in quanto si azzerano sempre dei bit con peso positivo. Ritornando all'esempio in cui si eliminano 4 bit il cui LSB ha peso 2^{-5} , per effettuare l'operazione di arrotondamento bisogna sommare al numero da troncare il valore 2^{-2} prima di eliminare gli ultimi 4 bit. L'errore che si commette varia tra -0.25 (quando i 4 bit eliminati sono '1000') a $0.2188 = 2^{-3} + 2^{-4} + 2^{-5}$ (quando i 4 bit eliminati valgono '0111'). Per le

Truncation

- Truncation in Verilog is a simple assign.
- Truncating signal A with 12bit to get signal B with 8bit is obtained with:

```
assign B=A[11:4];
```

Rounding

- Rounding is conveniently conducted into two steps. The addition of the rounding constant followed by the truncation operation
- The following code rounds a 16bit signal, B, to obtain the 8 bit signal A.

```
wire [7:0] A;              // Rounded signal
wire [15:0] B, tmp;        // Original signal and temporary signal
localparam [15:0] RoundConst=16'b0000_0000_1000_0000;
// 16'b0000_0000_1000_0000 is the rounding constant
// the _ symbol is ignored, it  is  included to improve readability

assign tmp = B + RoundConst; // Sums the rounding constant
assign A = tmp[15:8];  // Name A the most significant part of the result
```

operazioni di estensione:

Sign extension

- When the format needs to be extended to the left sign extension is needed
  - Zero padding if number is positive
  - One padding if number is negative
- Q3.0 -> Q6.0
  - 0111 -> 0000111
  - 1001 -> 1111001

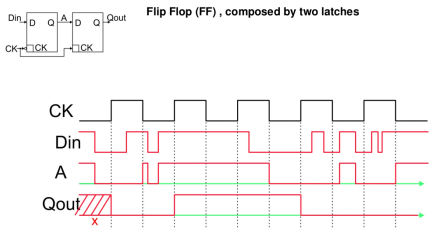
5.6 Moltiplicazioni TBD

Chapter 6

Circuiti sequenziali

6.1 Introduzione

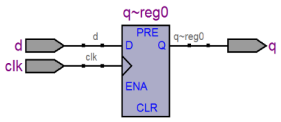
Circuiti che dipendono anche dal loro stato precedente. Non conviene implementare un circuito sequenziale tramite logica combinatoria, per esempio creando un flip flop dichiarando due inverter e poi collegandoli, ottenendo un **loop combinatorio**, i quali non vengono risolti né in sintesi né in simulazione. Utilizzeremo i blocchi procedurali con gli *always* block.



Prima dell'accensione, il rumore elettronico determina lo stato di Q_{out}, il quale si stabilizza una volta iniziato il ciclo. Il flip flop varia il suo stato sul fronte di salita (o di discesa) del clock, ergo nel blocco procedurale ci sarà la sola dipendenza da CK

Positive edge triggered D flip flop without reset/preset or enable

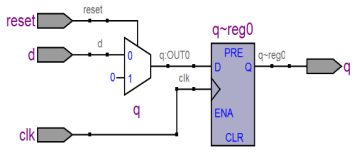
```
module df (d,q,clk);  // Module definition
input wire d,clk;
output reg q; // Assigned in a procedural block -> reg

always @(posedge clk)  // Procedural block.
       // Activated by the positive edge of the clock
begin
q <=d;  // D flip flop - non blocking assignment
end
endmodule
```



Una versione più completa del FF introduce il segnale di Reset, che blocca l'uscita a zero. O lo si fa **sincrono** oppure **asincrono**, dunque due circuiti diversi. Nel primo caso, l'effetto del reset lo vedo solo appena il clock si alza, nell'altro caso l'uscita viene cambiata istantaneamente.

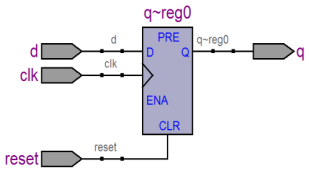
Positive edge triggered D flip flop with synchronous reset
Below the result of the synthesis on FPGA. The synchronous reset is implemented with a mux on the D signal.

```
reg q;  // Assigned in a procedural block => reg type
// positive edge triggered FF with synchronous reset
always @(posedge clk)
begin
   if (reset)
        q<=1'b0;
   else
        q<=d;
end
```



Positive edge triggered D flip flop with asynchronous reset
Below the result of the synthesis on FPGA. The asynchronous reset is yet present in the available FF and is directly exploited.

```
reg q;  // Assigned in a procedural block => reg type
// positive edge triggered FF with asynchronous reset
always @(posedge reset, posedge clk)
begin
   if (reset)
        q<=1'b0;
   else
        q<=d;
end
```



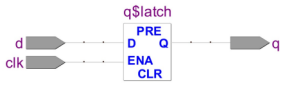
È importante usare la notazione blocking perché mettendo il minore uguale i due valori di q commutano istantaneamente, mentre il non-blocking segna i valori e li cambia soltanto quando arriva all'istruzione end.

It is uncommon to use latches in a well conceived circuit. The latches are usually the result of a wrong description of the circuit.

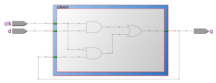
If a latch is really needed the description is that of a procedural block without the default statement and without describing the behavior in every case.

D latch transparent for clk = '1'

```
module latch_d (d,q,clk);
input wire d,clk;
output reg q;

always @(clk,d)
begin
if (clk)
        q <=d;
end
endmodule
```



RTL view



Technology map view

6.2 Test bench per circuiti sequenziali

A differenza del test bench per i combinatori, per i sequenziali bisogna implementare il clock in un blocco procedurale senza sensitivity list

```
localparam period=20; //Clock period is 20ns
    always
        begin
            CLKtb=1'b0;
            #(period/2.0); //2.0 is needed to obtain a real number
            CLKtb=1'b1;
            #(period/2.0);
        end

```

Il test bench per un sequenziale si articola in tre momenti:

- La simulazione di un circuito sequenziale implica prima di tutto la fase di reset. È buona pratica che, anche durante la fase di reset, tutti i segnali di input abbiano valore definito. Segue che la parte iniziale del test vector deve contenere le assegnazioni dei segnali di reset per tutti i segnali d'ingresso.

- La seconda cosa da fare è disattivare il segnale di reset. Per ragioni di leggibilità, ogni volta che un segnale viene modificato, è conveniente ridefinire tutti i segnali di input e dunque è buona pratica riassegnare il valore di ogni segnale al momento della disabilitazione del segnale di reset.

- Durante il terzo momento il segnale di reset è disabilitato, dunque variano soltanto i segnali d'ingresso. È consigliato, sempre per ragioni di leggibilità, di evitare di riassegnare il valore del segnale di reset durante le

rimanenti parti della simulazione, durante le quali ci limitiamo a cambiare soltanto il valore dei segnali d'ingresso.

Analizziamo un esempio di assegnazione corretta degli input e del segnale di reset. Il circuito d'esempio ha input A e B su 7 bit, un segnale di controllo $ctrl$ su 3 bit e i segnali di reset e clear (CLR) espressi su singoli bit. È assunto che il segnale di cler sia attivo basso e che i flip-flop siano sincronizzati sul fronte di salita del clock, generato dal codice scritto sopra.

```
initial
    begin
    //FASE 1: Reset attivo e valori di default per gli ingressi
    CLR = 1'b0; //Attivazione del reset
    A=y'b111_0000; B=7'b0; ctrl=3'b000;
    // Assegna un valore ad ogni input.
    //E' buona pratica non simulare con segnali indefiniti
    #(5*period) //Il reset dura cinque periodi di clock
    #(3*period/4.0); //Opzionale
                     //Le transizioni di input cominciano ad
                     //1/4 di periodo dopo il fronte di
                     //attivazione, il tempo per il quale
                     //e' permesso il ritardo logico
                     //combinatoriale e' 3/4 di periodo
    //FASE 2: disattivazione del reset
    CLR=1'b1; //Reset disattivato

    A=7'b000_0000; B=7'b0; ctrl=3'b000;
    // non abbiamo modificato solo A bensi' anche quelli che
    // non variano (buona pratica)
    #period;

    A=7'b000_0000; B=7'b0; ctrl=3'b001;
    #period;

    A=7'b000_0000; B=7'b101_1111; ctrl=3'b001;
    #period;

    A=7'b000_0100; B=7'b101_1111; ctrl=3'b000;
    #period;
    $stop; // sys-call per terminare la simulazione
end

```

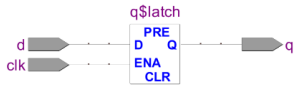
6.3 Altri circuiti sequenziali

6.3.1 D-latch

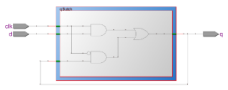
Non è cosa comune utilizzare un latch in un cicuito ben progettato, perché di solito questi sono il risultato di una descrizione errata. Se un latch è veramente necessario, la descrizione è quella di un blocco procedurale senza il default statement e senza descrivere il comportamento in ogni caso.

D latch transparent for clk = '1'

```
module latch_d (d,q,clk);
input wire d,clk;
output reg q;

always @(clk,d)
begin
if (clk)
        q <=d;
end
endmodule
```



RTL view



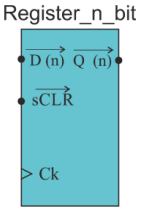
Technology map view

6.3.2 Registri

Un registro è un insieme di flip-flop connessi in parallelo, dunque la relativa descrizione in verilog sarà un flip-flop che prende in ingresso un vettore dando in uscita un vettore:

It is straightforwardly described as a Flip flop with arrays as input/output

```
module Register_n_bit (d,Ck,sCLR,q);
input wire Ck, sCLR; input wire [7:0] d;
output reg [7:0] q;

always @(posedge Ck)
begin
   if (sCLR)
      q<=8'b00000000;
   else
      q<=d;
end
endmodule
```



La versione con l'enabler si ottiene con un blocco if: $en?q <= d : q;$

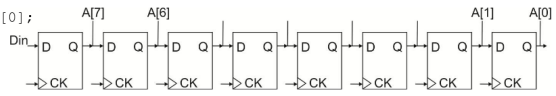
6.3.3 Registri a scorrimento

I registri a scorrimento (shift register) sono comunemente usati per conservare una sequenza di dati in ingresso. Il circuito top level mostrato in figura ha un bit d'ingresso ed n "taps", cioè dati conservati. Un utilizzo comune degli shift

▲ Or the whole set of stored data can be forwarded to the output.

register è quello di ritardare un segnale di n colpi di clock, dove n è il numero di tap del registro.

```
module shift_reg_8_taps_serial_out(Din,Dout,Ck,Reset);
input wire Ck,Din,Reset;
output wire Dout;
reg [7:0] a;  // internal nodes

always@(posedge clk)
begin
if (reset)
        a<=8'h00;
else
        begin // lines (a) and (b) can be exchanged
        a[7]<=Din;        // (a)
        a[6:0]<=a[7:1];   // (b)
        end
end
assign Dout=a[0];
endmodule
```

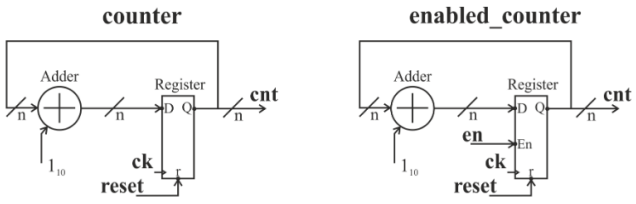


L'implementazione di sopra si può generalizzare al caso di uno shift register con ingressi ad m bit. Di seguito è riportato come esempio l'implementazione di un registro a scorrimento con ingresso a 16 bit ed 8 tap.

```
module shiftreg16b8tap(ck,reset,Din,Dout);
input wire ck,reset; input wire [15:0] Din;
output wire[15:0] Dout;
reg [15:0] D [7:0]; // flip flop array
integer k; // integer variable used in the FOR loop
always @(posedge ck, posedge reset)
begin
if (reset)
       for (k=0; k<8 ; k=k+1) D[k] <= 16'h0000; // loop
else
       begin
       for (k=7; k>0 ; k=k-1) D[k]<=D[k-1]; // loop
       D[0]<=Din; // outside the loop
       end
end
assign Dout=D[7];
endmodule
```

6.3.4 Contatori

I contatori sono parecchio comuni nei circuiti digitali. Sono usati per sincronizzare differenti sezioni in termini di numero di colpi di clock. Trovano svariate applicazioni: dividere bit in byte/word, determinare i bit di parità, permettere il time delay di segnali Quando non specificato altrimenti dal segnale di controllo, un contatore può contare da 0 a $2^n - 1$.
La struttura basica di un contatore è quella di un accumulatore, il quale a sua volta è composto da un addizionatore e da un registro. Contatori più complessi utilizzano segnali per contare fino a valori diversi da $2^n - 1$, generati aggiungendo strutture MUX.




```
module counter(ck,reset,cnt);
input wire ck,reset;
output reg [7:0] cnt;

always @(posedge ck)
begin
if (reset)
       cnt<=8'b00000000;
       else
       cnt<=cnt + 1'b1;
end
endmodule
```

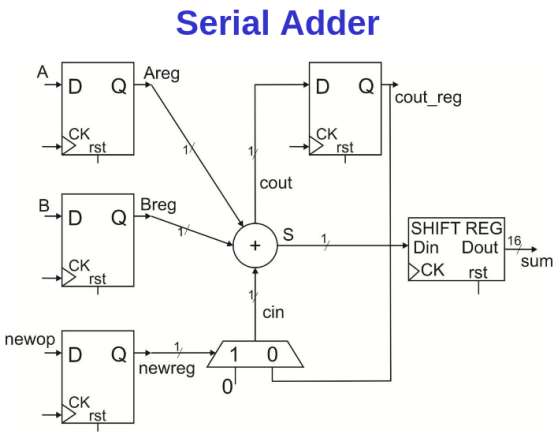
Per implementare una versione con limite superiore diverso da $2^n - 1$, basta aggiungere un altro blocco if che resetta il contatore una volta raggiunto tale limite.

6.3.5 Accumulatori

È come un contatore, solo che al posto di incrementare l'output di uno ad ogni colpo di clock, incrementa l'output aggiungendo il valore provvisto dall'ingresso.

6.3.6 Serial adder

Un addizionatore carry ripple è composto da full adder in cascata ed esegue la somma in un singolo colpo di clock. Se si ammette che tale operazione possa essere fatta in più di un colpo di clock, si può pensare ad una topologia che utilizza gli stessi full adder per fare una singola somma, risparmiando una notevole quantità di area. Per fare ciò bisogna passare i bit degli addendi uno alla volta, dunque un adder seriale prende dati d'ingresso da due registri e li somma. Il risultato va in un registro di uscita e viene shiftato di una posizione ad ogni somma che viene fatta. Il carry out va in un flip-flop e viene multiplexato con un segnale newop (nuova operazione): se si sta ancora facendo l'operazione, viene sommato il resto, altrimenti viene settato il carry in a zero e si riparte da capo.



Chapter 7

Temporizzazione di circuiti sequenziali

7.1 Circuiti sincroni e asincroni

I circuiti sincroni sono temporizzati tutti con lo stesso clock, mentre quelli asincroni no. Il 99% dei circuiti digitali sono sincroni.

Vantaggi dei circuiti sincroni
- L'analisi è semplice poiché il circuito si comporta come un sistema a tempo discreto.
- Progettazione altrettanto semplice.
- Affidabilità. Deriva dalle due proprietà elencate prima.
- Possibilità di testare velocemente il funzionamento del circuito finale.

Svantaggi dei circuiti sincroni
- Tutte le componenti del circuito elaborano alla velocità permessa dal circuito più lento.
- Il segnale di clock è difficile progettare poiché ha una grossa capacità di carico distribuita lungo tutto il chip e connette molti punti del circuito.

Vantaggi dei circuiti asincroni
- I circuiti possono andare alla massima velocità permessa del circuito stesso e dai dati in ingresso (esempio tipico è l'addizionatore per il quale gli ingressi che causano il massimo ritardo sono estremamente rari).
- Non serve progettare un segnale di clock globale.

Svantaggi dei circuiti asincroni
- Molto complesso da progettare.
- L'affidabilità dipende dai ritardi minimi e massimi di ogni singolo circuito.
- La verifica del corretto funzionamento (testing) del circuito finale è molto difficile.

7.2 Timing dei Flip-Flop

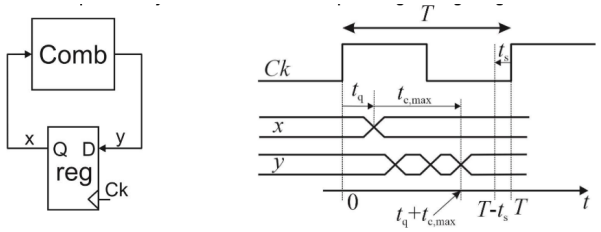
I flip flop si comportano come circuiti di campionamento. Il campionamento ideale ed il cambiamento istantaneo del segnale non è possibile. I parametri che determinano il comportamento di timing del flip flop sono:

- Il tempo di setup t_s, intervallo di tempo prima del fronte del clock in cui il dato deve essere stabile

- Il tempo di hold t_h, intervallo di tempo dopo il fronte di clock in cui il dato deve essere stabile

- Il tempo di ritardo del clock a q t_q, il ritardo di propagazione fra il fronte attivo del clock e il segnale q, cioè quanto ci mette il FF a variare la sua uscita.

Notare che t_q e t_h rappresentano due cose diverse: il primo è il tempo necessario affinché lo stato si propaghi in uscita, mentre il secondo il tempo necessario per evitare fluttuazioni di stato del FF prima di aver finito di caricare le capacità d'ingresso del circuito a valle. Vediamo un esempio di timing corretto e scorretto e vediamo in cosa differiscono. Partiamo dal diagramma temporale di un flip-flop e analizziamone i vincoli di setup



Required time (t_{req}): T-t_s

Arrival time (t_{arr}): t_q+$t_{c,max}$

Setup constraint: $t_{arr} \leq t_{req}$ -> $t_q+t_{c,max} \leq T-t_s$

With respect to clock period: $T \geq t_q+t_{c,max}+t_s$

Nello schema y è l'ingresso e x l'uscita[1] L'intervallo di tempo $T_{C,max}$ è il **ritardo combinatoriale massimo** e dipende dal percorso del segnale e deal numero di porte logiche attraversate dal circuito combinatorio. Come da definizione, il tempo impiegato da q (in questo caso chiamato x) per commutare è proprio t_q

Facciamo un quadro di ciò che sta accadendo in figura: l'ingresso y varie e dopo t_q secondi varia anche lo stato Q del flip-flop, rappresentato dal segnale x in figura. Poi il segnale x entra nella logica combinatoria e dopo un tempo combinatoriale che è al massimo $T_{C,max}$ questo esce. Il dato in uscita dalla logica combinatoria deve essere stabile almeno t_s secondi prima del prossimo fronte attivo (in salita nel nostro caso).
Questo ci porta a definire il primo dei due vincoli, quello di **setup**. Il vincolo di setup si calcola fra i due fronti attivi del clock ed è riferito a due circuiti che si passano il dato. Quello che eroga il dato è detto "launching" mentre quello che lo prende in ingresso è il "capturing". Nel disegno di sopra il FF è al tempo stesso launching e capturing, perché lancia il dato e se lo riprende dopo che questo è uscito dalla logica combinatoria. Se $t = 0$ corrisponde all'istante in cui c'è il primo fronte attivo e T quello in cui c'è il secondo, il vincolo di setup è

[1] Scelta peculiare

dato da:

$$T > t_q + T_{c,max} + t_s \tag{7.1}$$

questo perché dopo il fronte il dato ci mette $t_q + T_{c,max}$ secondi ad uscire dalla logica combinatoria e deve essere pronto almeno t_s secondi prima del prossimo fronte, dunque l'intervallo T deve essere abbastanza largo per soddisfare questo margine. La differenza fra T ed il valore minimo di T per cui il vincolo di setup è verificato e è detta **slack**:

$$\mathrm{T} = T_{C,max} + t_q + t_s + \mathrm{Slack} \implies \mathrm{Slack} = T - T_{c,max} - t_q - t_s \tag{7.2}$$

Dunque il vincolo di setup da un limite superiore alla frequenza di clock.

Passiamo ora al vincolo di **hold**, che si applica su un singolo fronte di clock. Il vincolo di hold è dato in sostanza dal tempo che serve al flip flop di far propagare il dato che ha già all'interno mentre arriva quello nuovo. In altre parole, è un vincolo su quanto lento deve andare il segnale in ingresso affinché lo stato precedente sia propagato senza fastidi.

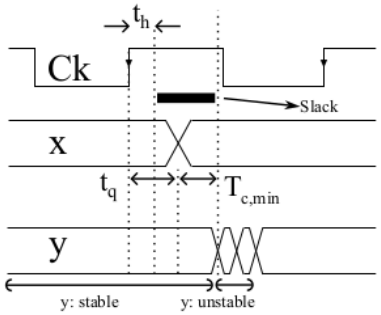


Figura 12.5: Vincolo sul tempo di hold.

Allora partendo dal fronte di salita devono passare t_h secondi dati dal tempo di hold prima che il dato nuovo arrivi. Tale dato impiega, per arrivare, un tempo minimo dato da $t_q + T_{c,min}$, dunque deve verificarsi:

$$T_{c,min} + t_q > t_h \tag{7.3}$$

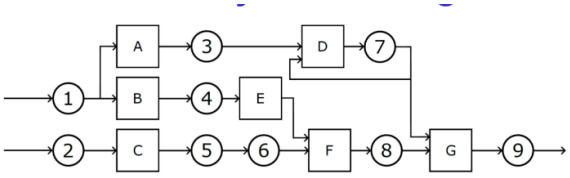
che come dicevamo prima è una condizione di "lentezza" per i ldato in ingresso. Anche qui si può definire uno slack:

$$T_{c,min} + t_q = t_h + \mathrm{Slack} \tag{7.4}$$

Per il vincolo di setup il progettista può procedere ad abbassare la frequenza, mentre per quello di hold si hanno le mani legate dalla tecnologia, perchè t_q e t_h sono caratteristiche del FF e $T_{c,min}$ della rete combinatoria, dunque a priori vanno scelti dei componenti che abbiano tempistiche compatibili. In ogni caso, un FF ben costruito ha il tempo di propagazione dello stato t_q maggiore di

quello di hold t_h.

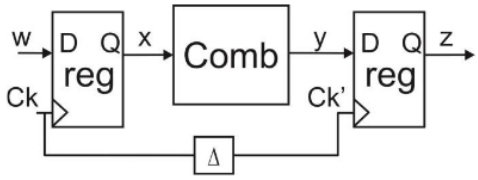
Per fare il timing di tutto il sistema e decidere la frequenza massima da utilizzare, si studia caso per caso il vincolo di setup e di hold e si raccolgono i valori di slack: il più basso determina la frequenza massima di lavoro. Se almeno uno degli slack ha valore negativo bisogna allungare il periodo di clock, abbassando la frequenza. Le espressioni riportate in figura discendono dalla



$1 \to 3$	$t_{q1} + T_{Amin} > t_{h3}$	$T > t_{q1} + T_{Amax} + t_{s3}$
$1 \to 4$	$t_{q1} + T_{Bmin} > t_{h4}$	$T > t_{q1} + T_{Bmax} + t_{s4}$
$2 \to 5$	$t_{q2} + T_{Cmin} > t_{h5}$	$T > t_{q2} + T_{Cmax} + t_{s5}$
$3 \to 7$	$t_{q3} + T_{Dmin} > t_{h7}$	$T > t_{q3} + T_{Dmax} + t_{s7}$
$4 \to 8$	$t_{q4} + T_{Emin} + T_{Fmin} > t_{h8}$	$T > t_{q4} + T_{Emax} + T_{Fmax} + t_{s8}$
$5 \to 6$	$t_{q5} > t_{h6}$	$T > t_{q5} + t_{s6}$
$6 \to 8$	$t_{q6} + T_{Fmin} > t_{h8}$	$T > t_{q6} + T_{Fmax} + t_{s8}$
$7 \to 7$	$t_{q7} + T_{Dmin} > t_{h7}$	$T > t_{q7} + T_{Dmax} + t_{s7}$
$7 \to 9$	$t_{q7} + T_{Gmin} > t_{h9}$	$T > t_{q7} + T_{Gmax} + t_{s9}$
$8 \to 9$	$t_{q8} + T_{Gmin} > t_{h9}$	$T > t_{q8} + T_{Gmax} + t_{s9}$

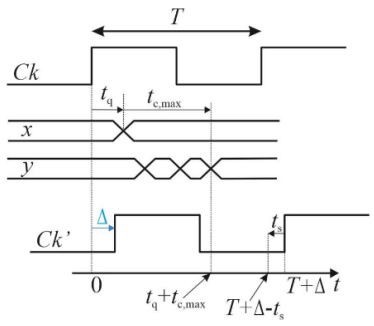
banale generalizzazione di quello di cui abbiamo parlato sopra. Se prendiamo ad esempio il pezzo di percorso $1 \to 3$, ci si rende conto che 1 lancia e 3 cattura, quindi il vincolo di setup deve tenere conto del tempo t_{q1} del lanciatore e del tempo di setup t_{s3} del catturatore, assieme al tempo combinatoriale massimo del blocco A. Per il vincolo di hold, è t_{h3} che pone il vincolo per il dato in ingresso, che per arrivare da 1 a 3 impiega $t_{c,min} + t_{q1}$.

A chiusura di questa sezione parliamo del Clock skew e gli effetti che questo ha sui vincoli di timing. Finora abbiamo considerato circuiti che hanno tutti lo stesso periodo di clock, ma possiamo pensare di inserire dei blocchi di ritardo del segnale per dare una versione ritardata del clock in ingresso ad alcuni blocchi del circuito, come in figura:



Il blocco di ritardo posticipa di Δt l'inizio dell'intervallo di setup del FF catturatore a destra, ma al tempo stesso ne riduce quello di hold della stessa quantità:

Se ricalcoliamo il vincolo di setup otteniamo, ipotizzando che i due flip flop siano



uguali:

$$T > t_q + T_{c,max} + (t_s + \Delta t) \tag{7.5}$$

Ma al tempo stesso, il vincolo di hold diviene:

$$t_q + T_{c,min} > t_h + \Delta t \tag{7.6}$$

questo perché traslando il clock si trasla anche l'istante per cui il tempo di hold è soddisfatto, visto che come riferimento t_h prende proprio il fronte attivo del clock. In sintesi, il clock skew può essere una risorsa nel caso si abbia molto margine con uno dei due parametri: per esepmio, lo slack del tempo di setup è molto ampio e invece di alzare la frequenza si vuole puntare sulla robustezza, allora si mette un blocco di ritardo da qualche parte per "donare" tempo di slack al vincolo di hold.

7.3 Applicazione al design di FPGA

Cenni agli alberi di clock

Di Flip-Flop all'interno di in un circuito digitale ce ne sono milioni, ragion per la quale pensare che il clock arrivi in sincronia ovunque è una brutale idealizzazione. Questo è dovuto, oltre al carico capacitivo non trascurabile di tali circuiti, anche alla lunghezza delle piste fra l'origine del clock e i flip-flop. Per ovviare a questa cosa, si è pensato di collegare i flip flop ad albero, interponendo dei buffer ogni tot sezioni. Le strutture di questi alberi sono calcolate tramite algoritmi appositi, fatti girare da programmi dedicati. Si ottiene una maggior omogeneità del clock skew in giro per il circuito, pagando però un grosso ritardo $T_{clk,buf}$ (legato ai buffer) tra l'origine dell'albero e i suoi punti capillari:

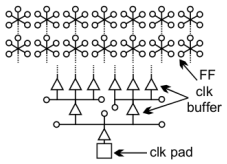
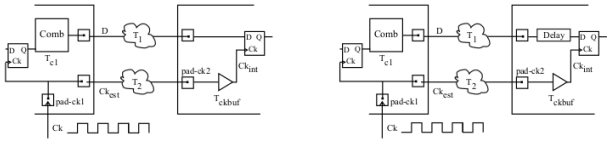


Figura 12.9: Schema dell'albero di buffer e collegamenti per la distribuzione del segnale di clock.

Pad to pad hold time

Per chi progetta su FPGA il clock tree non è una preoccupazione. Il progettista deve occuparsi però della scelta opportuna della frequenza di lavoro, perché deve soddisfare sia i tempi di hold e di setup all'interno del circuito che quelli dovuti all'interazione con circuiteria esterna, che lavora con tempi generalmente diversi.
Fino ad ora abbiamo ignorato la circuiteria legata al clock e assunto che le equazioni sui vincoli potessero essere ricavate dai tempi dei segnali in ingresso al flip-flop. Questa assunzione non è sempre corretta perché bisogna fare riferimento ai punti di circuiteria accessibili al progettista, perché è lì che si possono fare le misure per i vincoli. Nell'immagine che segue confrontiamo una situazione ideale con una reale, che tiene conto del ritardo dal buffer del clock che fornisce il ritardo $T_{clk,buf}$. Il ritardo di linea è dato da T_1 e T_2. Possiamo arbitrariamente inserire un blocco "Delay" di ritardo sulla linea dati:



(a) Circuito convenzionale. (b) Ritardo introdotto sulla linea dati.

Figura 12.10: Tempo di setup e tempo di hold visti dall'esterno di un chip.

Sotto queste condizioni la condizione di hold diventa:

$$t_{q,1} + T_{c,min} + T_1 + \text{Delay} \geq T_2 + T_{clk,buf} + t_{h2} \tag{7.7}$$

questo perché al tempo di transizione di stato t_{q1} va aggiunto il ritardo di linea, il ritardo del dato (blocco inserito da noi) e quello minimo della rete combinatoria. La somma di questi tre contributi deve essere maggiore del tempo di hold, a cui però va aggiunto il ritardo di clock e quello della rete di sotto T_2. Assumendo $T_1 = T_2$ e che i flip flop siano uguali si ottiene una relazione indicativa ma abbastanza accurata per il vincolo di hold:

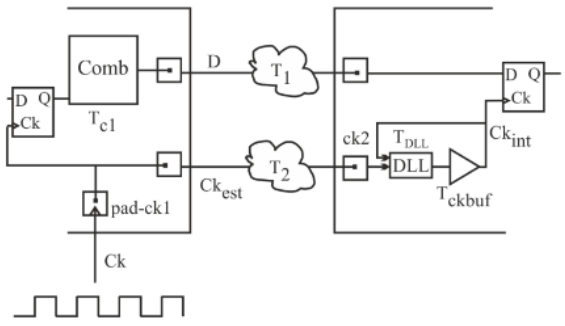
$$t_q + T_{c,min} + \text{Delay} \geq T_{clk,buf} + t_h \tag{7.8}$$

Il blocco di delay è arbitrario e lo si può scegliere in modo da essere comparabile al ritardo dovuto al buffer, ottenendo:

$$t_q + T_{c,min} \geq t_h \tag{7.9}$$

Aggiunta di un Delay Locked Loop

La situazione può essere migliorata ulteriormente per quanto concerne il tempo di hold aggiungendo un DLL (Delay Locked Loop) come nello schema in figura: In questo caso non abbiamo inserito il blocco di delay, bensì un blocco DLL



retroazionato che stabilizza il ritardo un ritardo sulla linea di clock dato da: $T_{DLL} = T_2 - T_{clk,buf}$. Assumendo sempre $T_2 = T_1 = T$ e FF uguali, il vincolo sul tempo di hold diventa:

$$t_q + T_{c,min} + T_1 \geq t_h + T_2 + (T_2 - T_{clk,buf}) + T_{clk,buf} \tag{7.10}$$

$$\implies t_q + T_{c,min} \geq T + t_h \tag{7.11}$$

Poiché il clock è periodico di T, la condizione si riduce a:

$$t_q + T_{c,min} \geq t_h \tag{7.12}$$

In altre parole, tramite il circuito di clock management basato su DLL possiamo far rispettare il vincolo di hold per due circuiti diversi che comunicano fra di loro in modo stabile e configurabile per diversi valori di ritardo di linea di clock.

Management tramite clock gating

La terza e ultima tecnica che vediamo per la gestione dei vincoli di timing è basata sulla tecnica del clock gating. Per clock gating si ci riferisce a quelle tecniche che utilizzano logica combinatoria per sincronizzare solo alcune parti del circuito. Anni fa, utilizzare il clock gating era un casino a causa degli ulteriori ritardi introdotti dalla logica circuitale e dal pericolo di generare glitch. Al giorno d'oggi sono indispensabili per azzerare la frequenza di lavoro delle parti inutilizzate del circuito e sincronizzare i ritardi. La loro robustezza è cresciuta molto grazie ai software che tengono conto dei ritardi nativi di questi elementi circuitali. Lo schema a cui facciamo riferimento è il seguente:

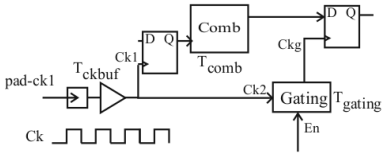


Figura 12.11: Circuito sequenziale che utilizza il clock gating.

Si vede già dallo schema che il clock gating infastidisce il tempo di hold, perché introdurrà un ritardo di natura combinatoria al clock che aumenterà il tempo di hold totale. Se partiamo dall'inizio, grazie all'albero di clock i due segnali Ck_1 e Ck_2 sono sincronizzati, quindi il paragone va fatto a partire da quei due punti li. Il tempo che ci mette il dato lanciato ad arrivare è $t_q + T_{c,min}$, mentre quello dato dall'hold sarà $t_h + T_{GATING}$. In definitiva:

$$t_q + T_{c,min} \geq T_{GATING} + t_h = t_{h,eff} \tag{7.13}$$

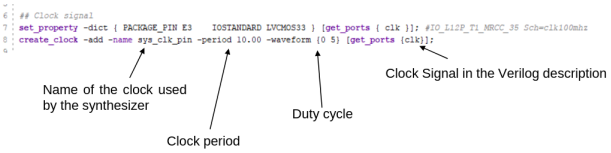
dove $t_{h,eff} = T_{GATING} + t_h$ è il tempo di hold "efficace". Il gating incide anche sul tempo di setup, che assume la forma seguente:

$$T > t_q + T_{c,max} + t_s - T_{GATING} \implies T > t_q + T_{c,max} + t_{s,eff} \tag{7.14}$$

che ha parecchio senso se si tiene conto del fatto che se peggiora l'hold a causa di un ritardo del clock migliora il setup, come abbiamo visto in precedenza.

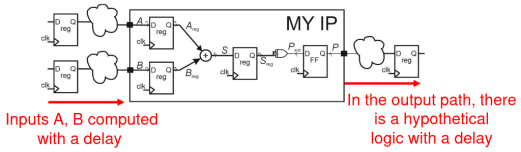
7.4 Timing analysis in vivado

Per il timing analysis in vivado, dobbiamo creare una risorsa .xdc per gestire il clock. In questo file dobbiamo essenzialmente dire a vivado di associare un clock virtuale all'ingresso di clock del nostro programma, così che possa verificare se i vincoli sono rispettati:

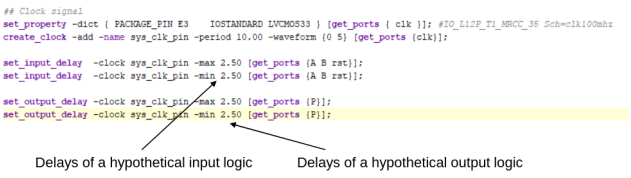


Questo comando cosa fa? Crea un oggetto clock (*create_clock*), lo aggiunge se non esiste già (-add) e gli da il nome di *sys_clk_pin* (*−namesys_clk_pin*), con periodo 10.00timescale (-period 10.00) con duty cycle del 50% (-waveform {0 5}) assegnandolo al segnale di clk dichiarato nel file verilog (il "programma"), tramite il comando [*get_ports*{*clk*}].

Concludiamo con un esempio. Consideriamo una situazione reale, dove ci sono due registri che comunicano un ingresso alla nostra IP (Intellectual Property, in pratica il circuito), che li elabora e li risputa fuori in pasto ad un altro circuito.



Dobbiamo tenere conto dei valori minimi e massimi di ritardo per i segnali di ingresso (A, B e il reset) e i valori minimi e massimi per quelli di uscita (P nel nostro caso). Allora il file .xdc si espande come segue:



Chapter 8

Dissipazione di potenza negli FPGA

8.1 Introduzione

La dissipazione di potenza è uno dei parametri più importanti da considerare nei circuiti digitali, perché costituisce un limite. Perché, in particolare, dovrebbe importarci di gestire la dissipazione di potenza?

- **Per l'utente mobile**: cellulari, fotocamere e qualsiasi altro device mobile ha una batteria, la quale si scarica molto più velocemente se la quantità di potenza dissipata non è ottimizzata
- **Potenza computazionale**: i circuiti caldi hanno problemi di saturazione di frequenza di lavoro e power throttling (letteralmente "strozzatura della potenza"), che ne riduce drasticamente le prestazioni
- **Inquinamento**: più consumi porta a più sprechi

Per il progettista, i punti 1 e 3 pongono dei limiti alla potenza media consumata dal circuito, mentre il punto due alla potenza di picco. Ridurre la potenza di picco consente di ridurre i costi e le dimensioni delle infrastrutture di raffreddamento nei chip che, a fronte della riduzione di potenza media, lavoreranno a temperature più basse. La riduzione della potenza media permette anche di avere alimentatori più piccoli e compatti. Tutto ciò migliora di gran lunga l'affidabilità del sistema.

8.2 Calcolo e stima della potenza

Come sappiamo, in un circuito digitale distinguiamo la potenza dissipata statica, legata soprattutto alle perdite, da quella dinamica, legata all'attività del

circuito.
Per quanto concerne la potenza dissipata statica, questa è data dal prodotto fra la corrente I_{DD} che il circuito assorbe dall'alimentazione e l'alimentazione V_{DD}. Poiché la corrente può dipendere dallo stato del sistema, se $P(\cdot)$ è la probabilità di stare in un certo stato:

$$P_s = V_{DD}[P(V_O = V_{OL})I_{DD}(V_0 = V_{OL}) + P(V_O = V_{OH})I_{DD}(V_O = V_{OH})] \tag{8.1}$$

Mentre quella dinamica è:

$$P_D = \alpha f_{clk} C V_{DD}^2 \tag{8.2}$$

dove C è la capacità di carico del nodo, f_{clk} la frequenza di clock ed α il numero di transizioni da $0 \rightarrow 1$ per periodo di clock. Un circuito con molti glitch ha $\alpha > 1$, mentre un nodo con la stessa uscita costante avrà $\alpha = 0$. A causa di ciò, α prende il nome di **switching activity**. In un circuito FPGA, la dissipazione di potenza dinamica è data dalla somma di tutte le potenze dinamiche dissipate sui nodi:

$$P_d = f_{clk} V_{DD}^2 \sum_{i}^{N_{\text{nodi}}} \alpha_i C_i \tag{8.3}$$

Fare questa cosa a mano è impensabile, per questo esistono dei tool appositi come vivado che hanno anche funzioni di Power Analysis. Per fare una stima corretta della potenza dissipata bisogna tener conto di sei parametri:

- La tensione di alimentazione V_{DD}, uniforme per tutti i blocchi
- La frequenza di clock f_{clk}, è decisa in fase di design con tutti i crismi del timing analysis
- La corrente $I_{DD}(V_{Oi})$ drenata dall'alimentazione verso il blocco i-esimo. Questo termine entra nel computo della potenza soltanto se il blocco in questione è effettivamente utilizzato nella sintesi circuitale. Il contributo che da è affidabile solo post-sintesi, cioè in fase di mapping.
- Le capacità C_i di carico per ogni blocco i-esimo che partecipa alla sintesi circuitale. Dipende dalle connessioni dei Logic Element (LE) del circuito. Il contributo che da è affidabile solo dopo la fase di Place and Route. Molto approssimativi nello stimare le capacità delle piste di connessione.
- Il parametro α_i per ogni nodo coinvolto, dipende dalla probabilità degli input di ogni LE. Varia in base allo scenario d'uso.
- La probabilità degli ingressi $P(V_{Oi})$, dipende dallo scenario d'uso.

Gli ultimi tre parametri variano in funzione del caso, dunque vanno fatte delle simulazioni con test vector.

A che punto del flusso di progetto fare le stime?

- In fase di **descrizione funzionale** è impossibile

- In fase di **descrizione RTL** è possibile abbozzare una prima stima molto molto grossolana
- In fase di **sintesi** si conoscono già i primi tre parametri della lista, mentre le capacità di carico possono essere stimane basandosi sulla lunghezza media delle piste di connessione all'interno dell'FPGA. Non avendo ancora fatto simulazioni, si possono assegnare valori standard di probabilità per includere il contributo degli ultimi due, per esempio supponendo che tutti i blocchi logici commutino una volta al secondo e abbiano in ingresso mediamente lo stesso numero di zero e di uno.
- In **simulazione post sintesi** si conoscono anche gli ultimi due parametri, la stima è quasi perfetta. L'unica cosa che manca sono i ritardi in simulazione che potrebbero causare glitch e di conseguenza cambiare le probabilità. Inoltre, la potenza statica dissipata è stimata in modo accurato solo se la simulazione combacia con lo scenario d'uso ed è processata per abbastanza tempo. Per intederci, se deve simulare una macchinetta del caffè è fuorviante fare il test per 500ns con dieci milioni di ingressi, se gli step che la macchina deve fare sono qualche migliaio (seleziona¿paga¿sorseggia).
- Arriviamo allora alla **Place and Route** nella quale si conoscono tutte le capacità di carico C_i, perché le connessioni dei blocchi sono fissati. Ora la stima della potenza dinamica dissipata è molto più precisa.
- In fase **post place and route** si conoscono completamente le probabilità e si ha una stima massimamente precisa.

In genere, la stima accurata in post place and route richiede un sacco di tempo, quindi quelle sulle potenza sono ottimizzazioni di second'ordine rispetto al timing, ma comunque importanti per quel che si è scritto.

Chapter 9

Macchine a stati finiti

9.1 Introduzione

Le macchine (o automi) a stati finiti sono essenziali nei circuiti digitali e si trovano ovunque. Sono rappresentabili come ASF i decoder, i riconoscitori di sequenze... . Come abbiamo visto per i sommatori, i flip flop e così via, anche gli ASF hanno le loro tecniche di descrizione dedicate nei vari HDL ed il Verilog che non fa eccezione. Questi strumenti vanno utilizzati per garantire che il codice sia leggibile da parte di chi lo consulta e ancora prima perché il sintetizzatore deve saper lavorare bene con quello che legge.

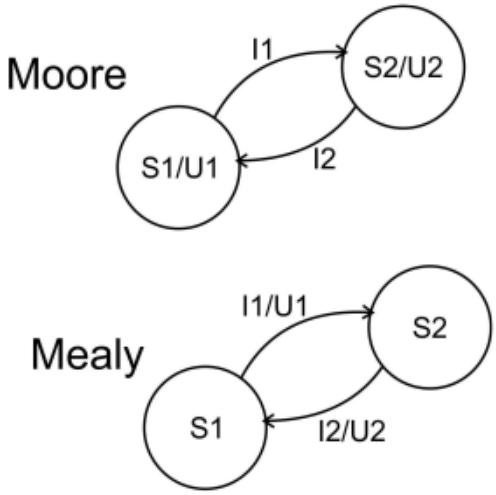
9.2 Cosa sono gli automi a stati finiti

Un automa è un sistema con le seguenti proprietà:

- **Dinamico**: evolve nel tempo passando da uno stato all'altro in funzione dei segnali d'ingresso e del suo stato precedente.
- **Invariante**: a parità di condizioni iniziali e di ingressi, si comporta allo stesso modo
- **Discreto**: le variabili d'ingresso, di stato e di uscita assumono solo valori discreti.

In ogni istante, un automa assume uno degli stati che compongono l'insieme degli stati. A fronte di un ingresso, l'automa subisce una transizione di stato e generando eventualmente (non necessariamente!) un messaggio d'uscita. Se l'insieme degli stati è finito, l'automa è a **stati finiti**. Dunque un ASF è un oggetto dinamico, invariante e discreto i cui possibili stati formano un insieme finito, simple as that. La controparte infinita riguarda la teoria della calcolabilità e a noi non interessa.

Come è intuibile, un'ASF si realizza tramite un circuito sequenziale, perché dipende dallo stato. Per realizzare un riconoscitore di sequenza, per esempio, servirà un circuito che tiene traccia degli ultimi n bit, compito svolto perfettamente da uno shift-register ad n tap.
Per rappresentare un automa ci si serve di un grafo orientato, i cui nodi sono gli stati e le frecce (archi) le transizioni da uno stato all'altro. Accanto agli archi si indicano gli ingressi che determinano le transizioni, mentre le uscite vengono indicate nel caso delle macchine di Moore all'interno di cerchi, mentre nel caso delle macchine di Mealy accanto agli ingressi. Nella realtà, scegliere



una macchina di Mealy o una di Moore può fare la differenza:

- Potrebbe essere più facile esprimere un problema in una rappresentazione rispetto che in un'altra
- Mealy risponde più velocemente, ma è più prona a presentare glitch
- Si potrebbe essere vincolati da standard di progetto
- Moore tende ad avere più stati di una macchina di Mealy

Gli automi a stati finiti hanno una limitazione: non esiste alcun ASF capace di prevedere se un dato programma finirà con un dato ingresso. Per esempio, non è possibile creare una macchina che dia in output $n + 1$ uno se gli viene data una serie di n zeri, perché dovrebbe funzionare per ogni n e ciò richiederebbe stati infiniti.
Le macchine a stati finiti all'atto pratico sono composti da un **banco di memoria** e da un blocco di **logica combinatoria**. Un banco di memoria da n bit può

reagire in 2^n differenti modi, dunque ci sono altrettanti possibili stati assumibili dall'automa. Se poi il tempo di lavoro dell'ASF è scandito da un clock, stiamo trattando con **macchine sincrone**.

Un ASF è definito a fronte di ciò che abbiamo detto come un insieme di elementi di memoria a due funzioni combinatorie F e G, tali che il prossimo stato è F(ingresso, stato corrente) e l'uscita è G(ingresso, stato corrente). In soldoni, per definire un'ASF occorrono due funzioni, una che determina lo stato e una che determina l'uscita. Per quanto concerne la funzione d'uscita, si ricava la **differenza fra Mealy e Moore**: in una macchina di Moore l'uscita dipende solo dallo stato corrente, mentre in una di Mealy sia dall'ingresso che dallo stato corrente.

Per progettare un automa a stati finiti, bisogna partire dalla descrizione funzionale, come sempre. L'iter è il seguente:

- Decisione delle specifiche: cosa deve fare la macchina? Conviene Mealy o Moore?
- Quali ingressi e quali uscite deve avere la macchina?
- Creazione del **bubble diagram**: viene individuato l'insieme degli stati e viene realizzato il grafo
- Tabella delle transizioni: Versione più completa del bubble diagram, comprende tutte le possibili transizioni
- Test dell'automa

Un esempio è riportato sul pacco di slide 8, pgg 19-23

9.3 Codifica dello stato

Il processo di codifica dello stato consiste nell'associare ad ogni stato una sequenza di bit da memorizzare nel banco, in sintesi la rappresentazione binaria di tutti i possibili stati del circuito che potrebbero avere significati più astratti. Un esempio con le macchinette del caffè:

- Caffè freddo $\rightarrow$ 000
- Caffè caldo $\rightarrow$ 001
- Caffè corto $\rightarrow$ 010
- $\cdots$
- Mocaccino $\rightarrow$ 111

e così via. Le possibili codifiche sono infinite perché sono infiniti i significati che uno può dare ad ogni codifica binaria. Dalla codifica dipende la complessità, la potenza dissipata e la velocità del circuito che implementa l'ASF.
In VHDL, il sintetizzatore può decidere la codifica di stato tramite l'assegnazione di un identificatore alfanumerico. In altri termini, si scrive direttamente "Caffè freddo", "Caffè caldo",... poi è il VHDL che decide quali simpatici numeretti assegnare. Nel verilog si utilizzano delle stringhe binarie da associare agli stati, dunque è il progettista a decidere. Ovviamente non è detto che si utilizzino tutti gli stati disponibili e conseguentemente ci saranno alcuni stati "orfani di significato". Questi vanno gestiti a parte perché fanno casino.

Ci sono vari approcci per la codifica. Il primo è la **one-hot**, che abbiamo visto nei primi capitoli quando abbiamo parlato dei decoder e degli encoder. Ogni stato ha associato una stringa con un solo 1 e tutto il resto zero. Dunque per n stati occorrono n flip-flop: lo stato "1" avrà un uno in prima posizione e il resto zeri, il "2" in seconda e così via. Una sua variante è la codica **zero-one-hot**, la quale per risparmiare un flip-flop associa lo stato con tutti zeri ad uno stato. In entrambi gli approcci, le condizioni nelle quali ci sono stringhe con più di un uno corrispondono ad errori. Tipicamente il segnale con tutti gli zeri viene associato al reset o di "partenza", perché resettando tutti i flip-flop a zero si casca proprio in quello stato.

Altro approccio è quello della **Codifica Gray**. Nel codice Gray il codice associato a due numeri consecutivi differisce sempre di un solo bit, per esempio $000- > 100- > 101- > 001$ e così via. Il Gray aiuta perché, nel caso il circuito abbia una probabilità elevata di attraversare stati consecutivi, il numero di commutazioni dei FF è minimo. L'esempio del caffè qui è istantaneo: Seleziona $(000)- >$ paga $(100)- >$gusta(101)[1]. C'è enorme vantaggio sulla potenza dissipata ed il rumore delle alimentazioni, inoltre sono necessari $\log_2(n)$ flip-flop per n stati. Un altro circuito tipico per questo tipo di codifica è il contatore.

Breve menzione va fatta anche alla **codifica Johnson**, simile a quella Gray, richiede $n/2$ flip-flop per n stati.

Stato	Codifica	Stato	Codifica
S_0	0000	S_4	1111
S_1	1000	S_5	0111
S_2	1100	S_6	0011
S_3	1110	S_7	0001

[1] seppur non sia uno stato proprio della macchina