

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

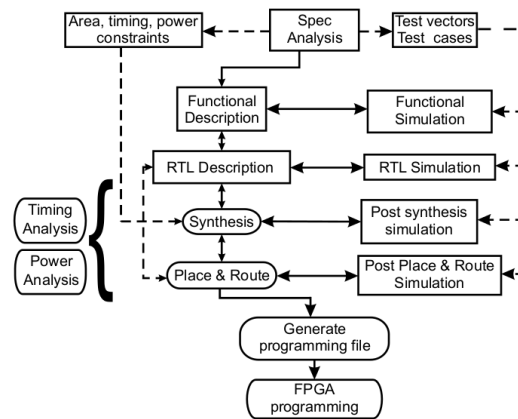


Figure 1.1: Flusso di progetto di un circuito digitale. Le fasi cerchiare in rettangolare le fa il progettista mentre quelle più rotonde sono semiautoatiche.

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 progettazione 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 constraint 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'iniezione 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.

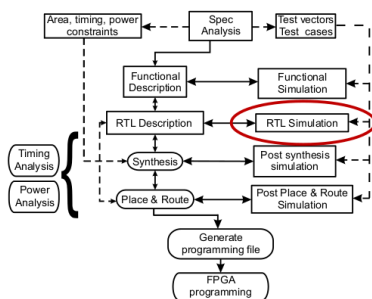
Le uscite desiderate utilizzate per controllare il funzionamento del circuito possono: essere calcolate manualmente; fornite dal committente; generate utilizzando la descrizione software del circuito ottenuta dalla fase di analisi delle specifiche.

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 necessari per compiere determinate 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 dà informazioni sui parametri del circuito quali potenza o area occupata, ma fornisce dati circa il throughput, la latenza e la sincronizzazione tra i blocchi.

▲ FFT example



- ▲ The circuit produces the first output after K clock cycles (latency)
- ▲ The circuit produces one output every N clock cycles (throughput)
- ▲ The circuit receives inputs in different clock cycles and needs to properly schedule the arithmetic operations

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 librerie 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 gating¹, questo si può simulare.

1.7 Place and route

Durante il place and route vengono piazzati (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 automaticamente.

¹In 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'utilizzo 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-

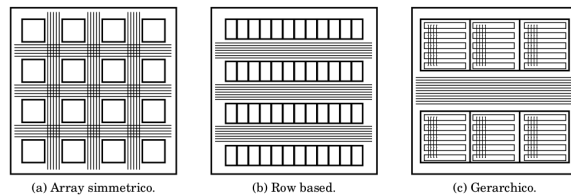


Figura 4.5: Possibili organizzazioni della logica e delle risorse di routing in un dispositivo FPGA.

trice e quella a righe sono alquanto 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.

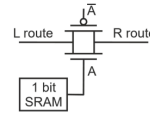
Le FPGA si possono distinguere anche a seconda del fatto che siano programmabili una sola o più volte, oppure che siano volatili e non:

	Volatile	Non-volatile
Repro-grammable	SRAM CMOS technology The programmable connection is big	Flash Flash technology The programmable connection is medium sized
OTP	NOT USEFUL	Anti-fuse Dedicated technology The programmable connection is small

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.

- ▲ A transmission gate connect L route with R route
- ▲ A 1-bit SRAM cell turns on or turns off the transmission gate
- ▲ Volatile and reprogrammable
- ▲ Simple CMOS process
- ▲ Large area and high resistive contact ($\sim k\Omega$)
- ▲ Slow critical path

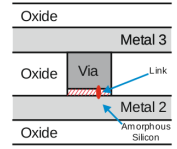


È necessaria una memoria flash esterna per conservare la configurazione, che viene caricata in memoria dinamica ogni volta.

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

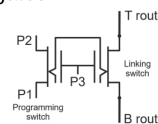
- ▲ Two metal regions with thin dielectric between them
- ▲ Programming voltage applied between metal regions
- ▲ The current flowing into metals and via creates the link
- ▲ Non-Volatile and OTP: once programmed, connection preserves over time and never changes
- ▲ CMOS process + further steps
- ▲ Contact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

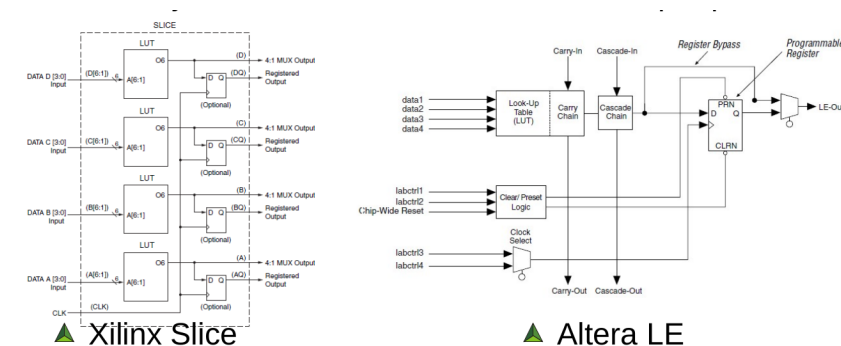
Un moderno FPGA si compone dei seguenti elementi:

- 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...

- Sulla parte periferica del chip FPGA troviamo:

- ## 2.3 Caratteristiche delle celle logiche programmabili

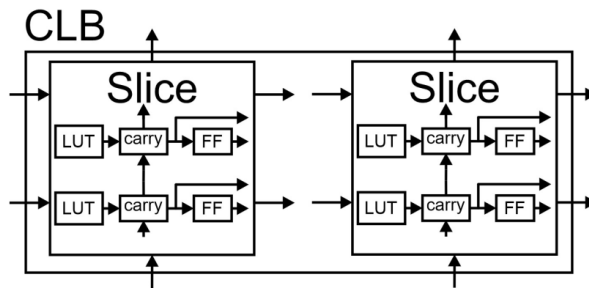
Generalmente gli LE implementano funzioni combinatorie. In alcune schede gli LE sono chiamati Slice. Dunque ogni Slice/LE ha delle LUT e dei flip-flop:



2.3. CARATTERISTICHE DELLE CELLE LOGICHE PROGRAMMABILI11

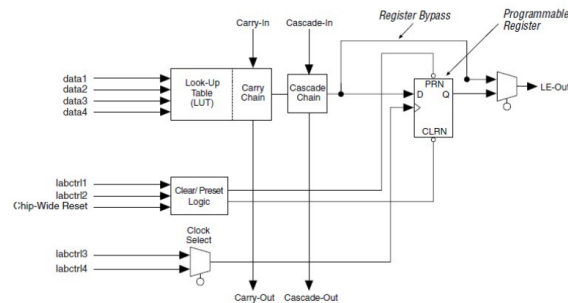
▲ Xilinx solution

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

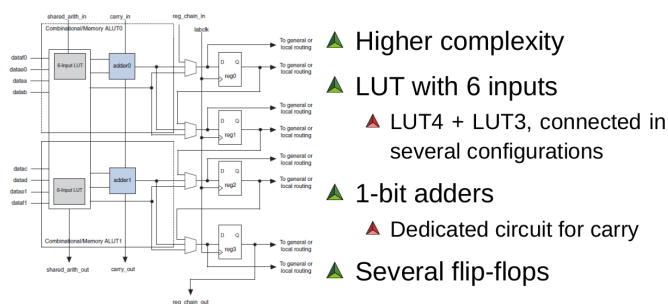


▲ Altera solution

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



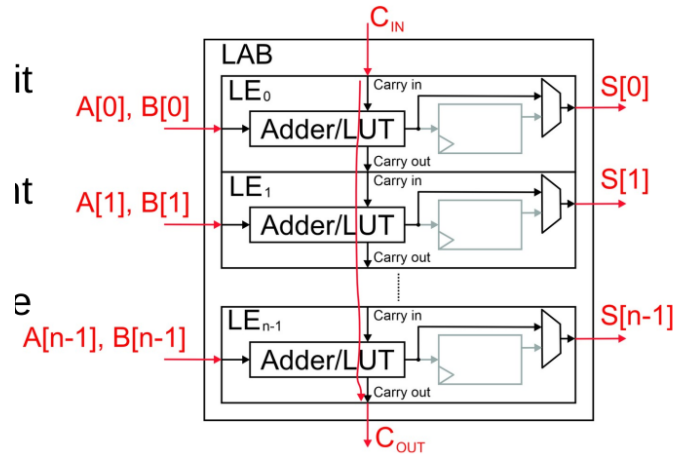
Le FPGA ad alte prestazioni implementano gli ALM (Adaptive Logic Module):



- ▲ 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 elementi 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,



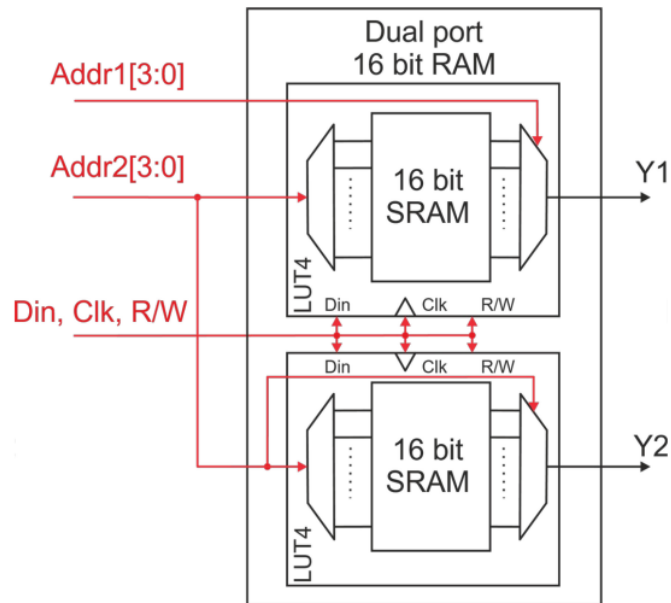
per garantire migliore flessibilità. Ci sono anche circuiti per la propagazione del riporto (carry-propagation), dunque connettendo n ALM in cascata è possibile creare addizionatori di $2n$ bit.

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.

2.3. CARATTERISTICHE DELLE CELLE LOGICHE PROGRAMMABILI13

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 prelevate in parallelo:

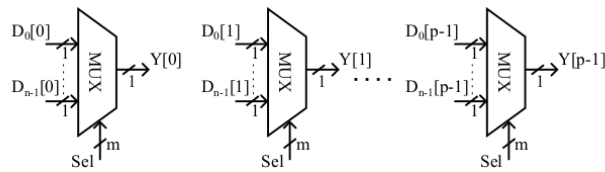


Figura 6.2: Il Mux con ingressi su più bit si realizza componendo Mux con un solo bit per ingresso.

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):

$$Y = A\bar{S} + \bar{A}S \quad (3.1)$$

cosciché 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

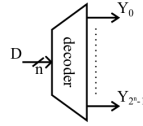


Figura 6.4: Il simbolo del decodificatore binario.

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

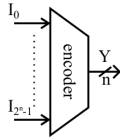


Figura 6.5: Il simbolo del codificatore binario.

I_0	I_1	I_2	\dots	I_{2^n-2}	I_{2^n-1}	Y (in decimal)
1	0	0	\dots	0	0	0
0	1	0	\dots	0	0	1
0	0	1	\dots	0	0	2
0	0	0	\ddots	0	0	\vdots
0	0	0	\dots	1	0	$2^n - 2$
0	0	0	\dots	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'encoder 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:

$$IsZero = \bar{I}_0 \bar{I}_1 \cdots \bar{I}_{n-1} = \overline{I_0 + I_1 + \cdots + I_{n-1}} \quad (3.2)$$

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:

$$Y = I_2 \cdot I_1 \cdot \bar{I}_0 = \overline{\bar{I}_2 + \bar{I}_1 + I_0} \quad (3.3)$$

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 :

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

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 a 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.

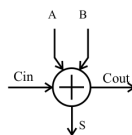


Figura 6.9: Simbolo e port di ingresso e uscita del full adder.

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 circuitale 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, continuous 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

decoder2to4

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

$$y_0 = \bar{a}_0 \bar{a}_1 \quad (4.1)$$

$$y_1 = a_0 \bar{a}_1 \quad (4.2)$$

$$y_2 = \bar{a}_0 a_1 \quad (4.3)$$

$$y_3 = a_0 a_1 \quad (4.4)$$

$$(4.5)$$

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

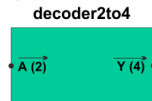
```

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 riutilizzati¹.

La keyword che consente di richiamare all'interno di un top level (circuito a gerarchia maggiore) un circuito lower level (un sottocircuito più elementare) è:

< *modulename* > < *instancename* > (< *interface ports* >)

Il modulo da mettere 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.

¹equivalente 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 stabilita all'interno del verilog con la keyword *posedge*):

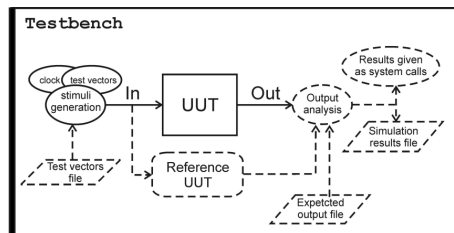
▲ Typical example of circuit whose unambiguous description is the key to let the synthesizers use the high performance flip flops yet present into the FPGA (or in the ASIC standard cell library).

```
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:

$$U_{n,m} \quad (5.1)$$

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:

$$Q_{n,m} \quad (5.2)$$

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

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

Per esempio

$$Q_{2,0} = 1010 = -2 \quad (5.4)$$

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).

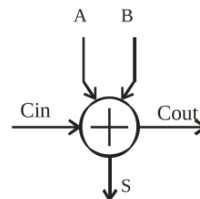
Si possono ottenere addizionatori a $2n$ bit mettendone in cascata 2 da n bit e collegando il C_{out} del primo al C_{in} del secondo. I riporti in entrata degli addizionatori al primo stadio sono posti a massa.

- ▲ 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

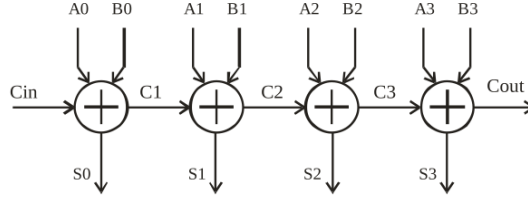
Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



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 addizionatore del tipo di sopra

▲ Delay :

$$T_{CR} \sim n * T_{Cin-Cout}$$



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:

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

5.4 Topologie di addizionatori per FPGA

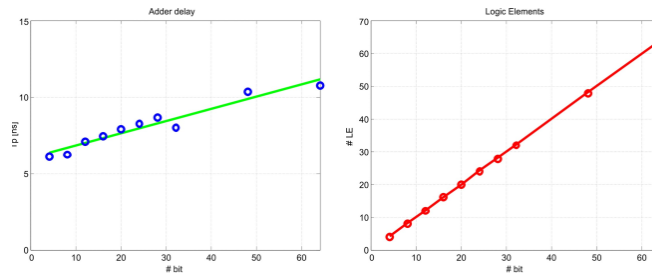
La topologia utilizzata per l'adder binario è quella a **carry ripple** con un piccolo fan-in. I blocchi logici delle moderne FPGA includono logica orientata alla propagazione del riporto, atta ad ottimizzare i collegamenti circuitali dei sommatore a carry ripple. Gli adder a carry ripple occupano molto 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:

$$OFL = C_{out,n-1} \oplus C_{out,n} \quad (5.6)$$

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:

Performance as a function of the number of input bits



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.

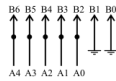


Figura 10.15: Estensione di un numero con due zeri meno significativi.



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 troncato 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

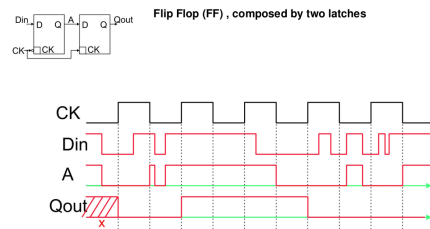
5.6 Moltiplicazioni TBD

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

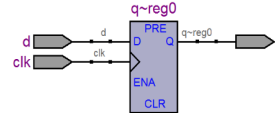
Positive edge triggered D flip flop without reset/preset or enable

```

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

```



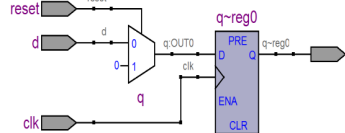
Una versione più completa del FF introduce il segnale di Reset, che blocca l'uscita a zero. O lo si fa **sincrono** oppure **asincrono**, dunque due circuiti diversi. Nel primo caso, l'effetto del reset lo vedo solo appena il clock si alza, nell'altro caso l'uscita viene cambiata istantaneamente.

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

```

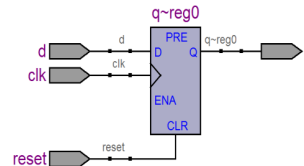


Positive edge triggered D flip flop with asynchronous reset
Below the result of the synthesis on FPGA. The asynchronous reset is yet present in the available FF and is directly exploited.

```

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.

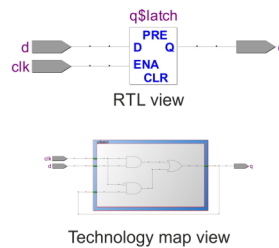
It is uncommon to use latches in a well conceived circuit. The latches are usually the result of a wrong description of the circuit.

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 $\text{clk} = '1'$

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

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



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

```
1 localparam period=20; //Clock period is 20ns
2 always
3 begin
4     CLKtb=1'b0;
5     #(period/2.0); //2.0 is needed to obtain a real number
6     CLKtb=1'b1;
7     #(period/2.0);
8 end
9
```

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.

```

1 initial
2   begin
3     //FASE 1: Reset attivo e valori di default per gli ingressi
4     CLR = 1'b0; //Attivazione del reset
5     A=y'b111_0000; B=7'b0; ctrl=3'b000;
6     // Assegna un valore ad ogni input.
7     //E' buona pratica non simulare con segnali indefiniti
8     #(5*period) //Il reset dura cinque periodi di clock
9     #(3*period/4.0); //Opzionale
10    //Le transizioni di input cominciano ad
11    //1/4 di periodo dopo il fronte di
12    //attivazione, il tempo per il quale
13    //e' permesso il ritardo logico
14    //combinatoriale e' 3/4 di periodo
15    //FASE 2: disattivazione del reset
16    CLR=1'b1; //Reset disattivato
17
18    A=7'b000_0000; B=7'b0; ctrl=3'b000;
19    // non abbiamo modificato solo A bensì anche quelli che
20    // non variano (buona pratica)
21    #period;
22
23    A=7'b000_0000; B=7'b0; ctrl=3'b001;
24    #period;
25
26    A=7'b000_0000; B=7'b101_1111; ctrl=3'b001;
27    #period;
28
29    A=7'b000_0100; B=7'b101_1111; ctrl=3'b000;
30    #period;
31    $stop; // sys-call per terminare la simulazione
32 end
33

```

6.3 Altri circuiti sequenziali

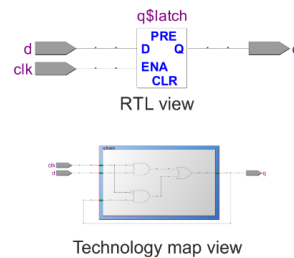
6.3.1 D-latch

Non è cosa comune utilizzare un latch in un circuito 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
```



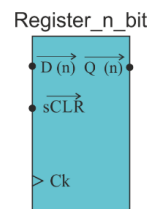
6.3.2 Registri

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: $en?q \leq d : q;$

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

```

A timing diagram illustrating the internal nodes of the 8-tap shift register. It shows eight D-type flip-flops connected in a chain. The input "Din" is connected to the D input of the first flip-flop. The Q output of each flip-flop is connected to the D input of the next flip-flop in the chain. The clock input "CK" is connected to the clock input of every flip-flop. The output "Dout" is taken from the Q output of the last flip-flop in the chain. The internal nodes are labeled A[7], A[6], A[1], and A[0] at the top of the flip-flop chain.

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.

```

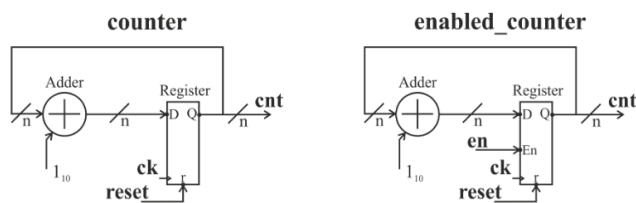
module shiftreg16b8tap(ck,reset,Din,Dout);
input wire ck,reset; input wire [15:0] Din;
output wire [15:0] Dout;
reg [15:0] D [7:0]; // flip flop array
integer k; // integer variable used in the FOR loop
always @(posedge ck, posedge reset)
begin
if (reset)
for (k=0; k<8 ; k=k+1) D[k] <= 16'h0000; // loop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

```

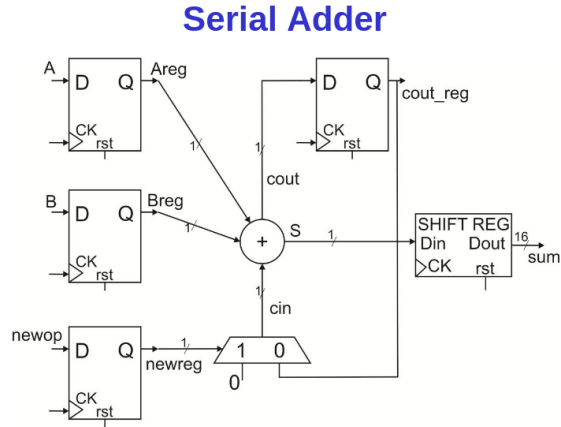
Per implementare una versione con limite superiore diverso da $2^n - 1$, basta aggiungere un altro blocco if che resetta il contatore una volta raggiunto tale limite.

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