Reti Logiche made Friendly

Matteo Sabella 218614

November 8, 2021

Chapter 1

Da analogico a digitale

Nell'era pre digitale i segnali erano analogici, ossia il loro valore spaziava su una gamma potenzialmente infinita di valori. Ciò poneva di fronte ad un problema, quello delle interferenze e dei disturbi di segnale, che potevano distorcere o rendere impossibile la lettura del messaggio trasportato dal segnale in questione. Si è perciò deciso di ricorrere al mezzo digitale, ossia un particolare modo di rappresentare le informazioni. Ogni segnale può assumere solo due possibili valori(da qui denominato segnale binario), che si distinguono grazie ad un confine detto soglia che li separa. Al di sotto di tale soglia il segnale ha ovunque lo stesso valore, così come al di sopra, ma tra una parte della soglia e l'altra, lì il valore cambia. Tanto più il segnale è lontano dalla soglia tanto più esso può subire distorsioni senza che il messaggio venga all'arrivo compromesso.

Tale sistema però pone uno svantaggio, ossia l' impossibilità di comunicare un vasto numero di messaggi. Per risolvere questo inconveniente intrinseco di questo modo di comunicare si ricorre a più sorgenti di segnali, così che invece che avere solo 2 possibili valori, il messaggio costituito da n segnali avrà 2^n possibili significati o sfumature.

Chapter 2

Circuiti combinatori

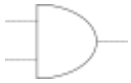
2.1 Operatori

2.1.1 OPERATORI SEMPLICI

Gli operatori basilari che vengono utilizzati per la costruzione di reti logiche sono i seguenti:

AND

L'operatore AND prende in input due segnali e restituisce 1 se e solo se entrambi sono 1, altrimenti restituisce 0.
Si rappresenta come : $x * y$



OR

L'operatore OR prende in input due segnali e restituisce 1 se almeno uno dei due è 1, altrimenti restituisce 0.
Si rappresenta come : $x + y$

NOT

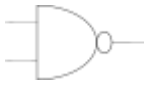
L'operatore NOT prende in input un solo segnali e restituisce 1 se il segnale è 0 e 0 se l'input è 1.
Si rappresenta come : x'

2.1.2 OPERATORI COMPLESSI

Gli operatori che vediamo di seguito possono essere considerati come operatori "complessi" ossia costituiti da elementi più semplici.

NAND

L'operatore NAND è costituito da un operatore AND seguito da un NOT.



NOR

L'operatore NOR è costituito da un operatore OR seguito da un NOT.

EXOR

L'operatore EXOR è anche detto OR esclusivo, questo perchè restituisce 1 se e solo se i due segnali in ingresso sono diversi tra loro.

EXNOR

L'operatore EXNOR è anche detto OR inclusivo perchè restituisce 1 se e solo se i due segnali in ingresso sono uguali tra loro.

2.2 Algebra di boole

Le operazioni svolte dagli operatori appena visti godono di proprietà che vanno sotto il nome di algebra di Boole. (Ognuna di esse è dimsotrabile tramite induzione completa, ossia verificando totalmente la veridicità delle implicazioni tramite tabelle di verità).

Proprietà distributiva

Complementazione

Proprietà associativa

Leggi di De Morgan

2.3 Teorema Di Shennon

Nel processo di ricerca di un'espressione logica che impieghi gli operatori che ora conosciamo, torna utilissimo il teorema di Shennon. Per ogni funzione logica del tipo

$f(x_1, x_2, x_3, ..., x_n)$

vale la seguente uguaglianza.
$f(x_1, x_2, x_3, ..., x_n) = a * f(1, x_2, x_3, ..., x_n) + a' * f(0, x_2, x_3, ..., x_n)$

Questa uguaglianza è valida perchè se a valesse 1, allora la parte di destra dell'OR sarà un AND con sicuramente valore 0 e quindi verrà come risultato il valore della prima espressione, altrimenti se a fosse 0 varrebbe lo stesso discorso ma al contrario.

Procedendo iterando questo teorema sui pezzi di funzioni rimanenti si può arrivare ad un'espansione nella seguente forma:
$f(x_1, x_2, x_3, ..., x_n) = a * f(1, x_2, x_3, ..., x_n) + a' * f(0, x_2, x_3, ..., x_n)$

Chapter 3

Semplificazione di funzioni logiche

Introduciamo ora alcune definizioni:

- **Letterale** Ogni variabile che sia affermat o negata.
- **Minterm** Prodotto in cui ogni variabile compare una volta come letterale.
- **Maxterm** Somma in cui variabile compare una volta come letterale.
- **Implicante** f è implicante di g se e solo se quando f è 1 allora g è 1.
  Si dice **implicante primo** ogni implicante che non è possibile racchiudere in uno più grande. Essi possono essere **ridondanti** nel caso in cui coprano zone coperte da altri implicanti primi mentre altri sono invece **essenziali** perchè sono gli unici a coprire quelle zone.
  Ogni minterm è un implicante della propria f, l'unico problema è che ognuno di essi rappresenta "pochi uni".

Per trovare implicanti che non siano minterm è utile utilizzare la **notazione Gray**. Essa consiste nel fare tabele di verità in cui tra ogni riga e la sua successiva c'è solamente una variabile a variare, così da poter eventualmente fare dei raccoglimenti sfruttando l'algebra di Boole.

Nel caso funzioni con più di due variabili è necessario cercare gli implicanti utilizzando altri modi per rappresentare le tabelle di verità così da far risaltare meglio le adiacenze tra i vari minterm.

Qui entra in gioco **la mappa di Karnough**

3.1 Insiemi di operatori completi ed uso delle porte logiche

3.2 Mappe a 5 e 6 variabili

3.3 Altri metodi per sintetizzare circuiti

3.4 Minimizzazione congiunta

Chapter 4

Circuiti Combinatori Fondamentali

Di seguito vengono rappresentati, analizzati e spiegati alcuni circuiti combinatori che sono ritenuti fondamentali in quanto compaiono molto spesso e per cui è bene ricordarsi come sono fatti per non perdere tempo.

4.0.1 Multiplexer

In generale un multiplexer è una funzione logica che tramite un criterio prende delle variabili in entrata e ne restituisce una di esse.

Da 2 a 1

Ora prendiamo in esame il più elementare tra i multiplexer, ovvero quello che fa una decisione tra due varibili in ingresso. Questo è il più elementare perchè altrimenti avrei una variabiule in ingresso ed una in uscita e ciò non è un vera scelta.

Partiamo dalla tabella di verità scritta con il codice Grey e cerchiamone gli implicanti primi. Partiamo da qui perchè è il punto principale per impostare come funzionerà il nostro multiplexer.

S-AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0

Gli implicanti primi sono: SB, AB, S'A. Quindi la nostra funzione può essere scritta come segue: f = SB+S'A

Cerchiamo di capire cosa ci sta dicendo questa funzione. Premettiamo che la convenzione che S=0 selezioni A mentre S=1 selezioni B è nostra scelta, con alcune differenze può avvenire benissimo anche il caso in cui S=0 selezioni B mentre S=1 selezioni A.

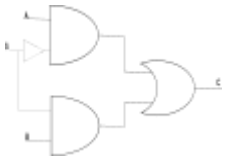


Figure 4.1: schema circuitale di un Multiplexer 2 a 1

Prendendo in esame uno solo delle AND, queste ci dicono che nel caso in cui S sia 0 SICURAMENTE quel termine sarà del tutto irrilevante nel passaggio successivo costituito da un OR. Perciò verrà selezionato ESCLUSIVAMENTE l'operazione AND che ha S=1. Poi l'essere 0 o 1 dell'altro operando determinerà l'uscita esattamente di quel valore.

Il multiplexer 2 a 1 si rappresenta con un questo simbolo circuitale: In cui i numeri indicano per quale valore di S venga selezionata la corrispondente entrata.

Da 4 a 1

Alla luce di quello che abbiamo appena letto riguardo al multiplexer 2 a 1 possiamo immaginare come realizzare questa versione più in grande.

Posso utilizzare due multiplexer 2 a 1 per ottenere due output e poi questi ultimi metterli all'interno di un ulteriore multiplexer 2 a 1 così da ottenere un solo risultato in uscita.

S_{01} si occuperà di scremare tra A e B mentre S_{02} scremerà tra C e D.

4.0.2 Odd Function

4.0.3 Conta Uni

4.0.4 Codifica e decodifica

Decodifica per display a 7 segmenti

Priority Encoder

4.1 Indifferenze o don't care

Chapter 5

Tempistiche e diagrammi temporali

In un circuito ci sono dei ritardi tra quanto il segnale entra in una porta e quando ne esce il risultato prodotto dalla logica della porta stessa. Questi ritardi sono dati da motivi fisici e di solito sono specificati al momento dell'acquisto della porta così da permettere di scegliere quella più adatta agli scopi per cui verrà impiegata.

I diagrammi temporali sono di due tipi:

SENZA RITARDO

Quelli senza ritardi vengono impiegati per valutare la funzione logica ed effettuare operazioni di debugging se necessarie.

CON RITARDO

Quelle con ritardi invece vengono utilizzati per valutare le prestazioni della funzione logica.

Un tempo utile da conoscere riguardo ad un circuito logico è quello massimo che può impiegare per restituire una risposta all'input sottopostogli.

5.1 Glitch

In un circuito possono verificarsi dei fenomeni detti **Alee o Glitch** e sono dovuti a brevi e rapidi cambiamenti di valore che avvengono con particolati combinazioni di ritardi. Influiscono sul risultato istantaneo ottenibile dal circuito (transitorio) ma non sulla correttezza della funzione logica.

Per ovviare a ciò si utilizza la tipologia di **circuito sincrono**

5.1.1 Alee su Karnaugh

Chapter 6

Circuiti aritmetici

Si chiamano **circuiti aritmetici** tutti quei circuiti che sfruttano porte logiche e il sistema di numerazione binario per effettuare operazioni di calcolo.

6.1 Struttura iterativa generica

6.2 Somma

Half adder

Iniziamo con la tabella di verità del circuito che dovrebbe sommare due ingressi da 1 bit ciascuno. Esso restituirà un risultato con 2 bit in uscita. Indichiamo con a e b i due ingressi, mentre con s il bit meno significativo del risultato e con c quello più significativo. La lettera c ci ricorderà anche tale cifra è dovuta al carry dell'operazione.

a	b	c	s
0	0	0	0
0	1	0	1
1	1	1	0
1	0	0	1

Mettendo s e c in due mappe di Karnaugh separate si possono costruire le funzioni logiche di ognuno. Iniziamo con S:

a/b	0	1
0	0	1
1	1	0

La funzione logica che rappresenta l'uscita s quindi è aXORb

Passando a C otteniamo poi:

a/b	0	1
0	0	0
1	0	1

La funzione logica che esprime il comportamento di C è aANDb

Il circuito dell'half adder è:

Full adder

Prestazioni

6.3 Sottrazione

6.4 Moltiplicazione

Chapter 7

Linguaggi di descrizione dell'hardware

Chapter 8

VHDL

Il linguaggio che utilizzeremo è il **VHDL**, ossia **V**ery highspeed integrated circuit **H**ardware **D**escription **L**anguage.

Il VHDL ci permette di lavorare secondo 3 **metodologie**

- **Top-Down**
- **Buttom-Up**
- **Meet In The Middle**

e secondo 3 viste

- **Data flow**:
  Consiste nella descrizione delle uscite in funzione degli ingressi. E' specificata con equazioni booleane messe a sistema tra loro.
- **Strutturale**:
  Consiste nella descrizione del circuito grazie all'utilizzo di componenti già esistenti che verranno combinati ed aggregati tra di loro.
- **Comportamentale**:
  Consiste nella descrizione del circuito tramite l'algoritmo che dovrà implementare.

8.1 Entità

La nozione di **entità** in VHDL corrisponde alla rappresentazione di un blocco **SENZA SPECIFICARNE** la logica interna.

Esso specifica :

- **nome**
- **numero di ingressi**
- **numero di uscite**

Di seguito il codice per dichiarare un'entità di nome $nom_entityconningressiem-nuscite$.

```
entity nome_entity is
port
(
nome_porta1 : tipo_ingresso,
nome porta2 : tipo_ingresso,...
nome_portan : tipo_ingresso;
nome_portan+1 : tipo_uscita,
nome_portan+2 : tipo_uscita,...
nome_portan+3 : tipo_uscita;
);
end  nome_entity;
```

8.2 Architettura

La nozione di **architettura** in VHDL specifica il comportamento di un'entità.
(Ad un'entità possono corrispondere più architetture.)

```
architecture nomearchitettura of nomeentità is
--dichiarazione di eventuali signal o buffer ausiliari
begin
--le equazioni e le operazioni di assegnamento che verranno scirtte in questa zona ver



end architecture nomearchitettura;
```

8.3 Testbench

Il **testbench** è un'unione della vista dataflow con quella strutturale che permette la simulazione del circuito dando dei dati in ingresso.

```
--dichiaro un'enetità senza interfaccia perchè non ha necessità di porte di ingresso nè di uscita

entity TestBench is
end entity TestBench;

--definisco ora la sua architettura

architecture test of TestBench is

--segnali interni di interconnessione
signal a,b,c : bit;

begin

--istanzio il modulo da testare
g1 : entitàdatestare portmap()

--definizione degli stimoli tramite equazioni in cui assegno ogni tot secondi un certo valore ai
```

Il testbench può essere fatto nello stesso file di testo in cui si dichiarano le entità e le architetture che poi andranno ad essere usate.

Components

Può capitare di avere una funzione logica implementata in un file e di volerla recuperare all'interno di un progetto più ampio o per eseguirne un test bench ma senza sporcare il codice.

Per fare ciò è necessario dunque all'interno del nuovo file pescare questa risorsa e ciò è possibile tramite un component.

```
component nomecomponent is

--descrizione dell' entità

end component;
```

Operatori,Identificatori e operazioni multiple

Il linguaggio VHDL ha di default tutti gli operatori booleani . L'operatore **NOT** è l'unico con la **priorità alta** mentre gli altri hanno tutti la stessa priorità e quindi verrà data importanza all'ordine o ad eventuali parentesi per decidere quale operazione fare per prima.
VHDL è un linguaggio NON case sensitive.

Per specificare un delay si può utilizzare l'operatore **after** Con il tipo bit_vector(n down to m) si crea un array di bit con gli indici che vanno da m ad n. VHDL possiede tutte le operazioni booleane come and,or,xor... La precedenza maggiore tra tali operazioni la ha la porta NOT mentre a differenza di quello che avviene nell'algebra booleana il resto degli operatori ha la stessa priorità perciò per forzare alcune operazioni prima di altre bisogna usare le parentesi !Nella descrizione delle architetture a volte è necessario instanziare variabili interne, queste vengono chiamate **signal** e necessitano di un nome univoco per essere riconosciuti nello svolgimento della logica del componente. (Attenzione che VDHL NON è case sensitive).

All'interno della stessa architettura possono avvenire più operazioni. L'ordine in cui vengono specificate all'interno dell'architettura non è rilevante perchè vanno concepite come un sistema in cui procedono in parallelo.

Parametri formali ed effettivi

8.4 Modi dei segnali

I segnali possono assumere vari tipologie e ognuna di esse comporta un certo set di operazioni ammesse o no.

- **IN** I segnali in in possono essere solamente letti e perciò possono stare solo a destra dell'operatore di assegnazione.
- **OUT** I segnali di out possono essere solo scritti e perciò staranno solamente dalla parte dell'operatore di assegnazione che subisce l'assegnamento. Inoltre nelle quazioni all'interno di un'architettura possono comparire una volta sola, questo perchè none ssendo sequenziali bensì contemporanee le istruzioni avrei un conflitto nell'assegnazione.
- **BUFFER** I segnali buffer sono segnali in out ma leggibili e perciò possono stare da entrambe le parti di un'assegnazione. Ma possono subire un'assegnazione comunque una sola volta.

  ```
  port(var : buffer bit);
  ```

- **SIGNAL** Sono segnali sia in ingresso sia in uscita (venendo valutati una volta sola).

  ```
  --si scrive nell'architecture PRIMA del begin
  signal S: bit_vector(3 downto 1);
  ```

- **INOUT** Sono componenti analoghi ad i buffer ma possono avere driver dall'esterno.

  Possono subire assegnazioni più volte ma stando attenti ai conflitti. (es. buffer tri-state)

  ```
  --in entity
  port (bus: inout bit)
  ```

- **GENERIC**

  Permette di parametrizzare un segnale così da rendere un'entità più versatile e riutilizzabile.

  nella definizione dell'entità si scrive

  ```
  generic(nomesegnale : tiposegnale)
  ```

  poi nella definizione dell'architettura di quella data entità si scriverà

  ```
  g1 : entità generic map(nomesegnale <= valore)
  ```

oppure si può usare la notazione posizionale nel caso di più generic per rendere tutto più compatto.

- **CONSTANT**

  Permette di definire costanti per leggere ed interpretare il codice più agevolmente e poi per facilitare eventuali modifiche.

```
--in architecture PRIMA di begin
constant nomecostante : tipo :=valore;
```

- **TIPI DEFINITI DALL'UTENTE**

```
--in architecture PRIMA di begin
type nomedeltipo is tipo
```

- **TIPI COMPOSITI**

enum	type nomeenum is (lista di valori che può assumere)
array	type nomearray is array (come sono gli indici) of tipodicontenuto
struttura	type nomestruttura is record (dichiaro insieme di variabili che lo costituiscono) end

8.5 Far girare il codice vhdl : ghdl e gtkwave

I programmi che andremo ad utilizzare sono ghdl e gtkwave.

Una volta scaricate le versioni stabili correnti adatte al vostro dispositivo si consiglia di estrarre i contenuti di entrambe le cartelle in due cartelle separate.

Poi aggiungere nel caso di windows i percorsi delle cartelle bin al PATH così da poterci accedere ogni volta senza dover andare nel percorso preciso in cui andremo a mettere gli eseguibili.

ghdl

gtkwave

Sporcarsi le mani

I comandi più frequenti sono:

```
ghdl -h
```

visualizzo i comandi che ghdl offre.

```
-a nomedelfile
```

Compila il file .vhd che gli indichiamo

`-e UNIT`

E' un passaggio necessario ai fini dell'esecuzione. Nel token UNIT va specificato il nome della entity che poi abbiamo intenzione di eseguire.

`-r UNIT`

Questo comando esegue l'architettura dell'entità che inseriremo al posto di UNIT e che prima dovrà essere passata per i comandi -a e -e.

Quindi se volessimo compilare ed eseguire l'entità baseEntity nel file baseTest.vhd seguiremo i seguenti passaggi:

```
ghdl -a baseTest.vhd
ghdl -e baseEntity
ghdl -r baseEntity
```

Tipo std_logic

Il tipo standard logic (std_logic) è un' estensione del concetto di bit che permette di rappresentare ulteriori stati logici.

U	Undefined
X	Non determinato
0	0 logico
1	1 logico
Z	alta impedenza
W	segnale debole non determinabile
L	segnale debole tendente ad 0
H	segnale debole tendente ad 1
-	don't care

Chapter 9

Macchine a stati