Reti Logiche made Friendly

Matteo Sabella 218614

January 12, 2022

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

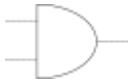
2.1 Operatori

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

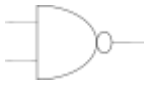
L'operatore NOT prende in input un solo segnali e restituisce 1 se il segnale è 0 e 0 se l'input è 1.
Si rappresenta come : x'

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).

Identità

$x + 0 = x$
$x * 1 = x$

Proprietà commutativa

$x + y = y * x$
$x * y = y * x$

Proprietà distributiva

$x * (y + z) = (x * y) + (x * z)$
$x + (y * z) = (x + y) * (x + z)$

Complementazione

$x + x' = 1$
$x * x' = 0$

Proprietà associativa

$x + (y + z) = (x + y) + z$
$x * (y * z) = (x * y) * z$

Leggi di De Morgan

Consentono di trasformare delle funzioni in cui compaiono OR in funzioni equivalenti che usano delle AND.

$(x + y)' = x' * y'$ $\qquad$ $(x' + y')' = x * y$
$(x * y)' = x' + y'$ $\qquad$ $(x' * y')' = x + y$

Tip : un modo utile per ricordarsele è che per passare da un operatore ad un altro , si cambia l'operatore e si "raccoglie" un not.

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 affermata o negata.

- **Minterm** Prodotto in cui ogni variabile compare una volta come letterale. Esso è 1 per una e una sola combinazione di letterali.

- **Maxterm** Somma in cui variabile compare una volta come letterale. Esso è 0 per una e una sola combinazione dei suoi letterali.

- **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".

La funzione logica sarà data dall' **OR** degli implicanti, ma quali cerco ?

Partendo dalla tabella della verità si possono trovare i minterm della funzione.
Questi poi possono essere compattati utilizzando le proprietà dell'algebra di Boole (torna molto utile l'adiacenza logica).

L'adiacenza logica in particolare si può utilizzare solo quando, dati minterm composti da L letterali, L-1 letterali restano immutati ed uno solo cambia.

Per trovare implicanti che non siano minterm è utile utilizzare la **notazione Gray**.
Chiamiamo **term** gli implicanti che non sono minterm !.
Essa consiste nel fare tabelle 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.

Non garantisce tuttavia un modo perfetto per trovare i minterm con facilità perchè permette di muoversi in una sola dimensione.

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**

a\ bc	00	01	11	10
0	*	*	*	*
1	*	*	*	*

Nella prima riga partendo dall'alto verranno messe in notazione Gray tutte le

possibili combinazioni che la variabile bc può assumere. Nella prima colonna invece tutte le combinazioni (in questo caso solo due) che la variabile a può assumere.

Nelle celle interne invece si procederà mettendo ad ogni incrocio, al posto di * l'uscita della funzione per i corrispondenti valori di a,b e c.

...

3.1 Insiemi di operatori completi ed uso delle porte logiche

3.2 Mappe a 4,5 e 6 variabili

Le mappe saranno fatte esattamente come fatto fino a qui con la differenza che aumenterà la loro dimensione in modo esponenziale. Infatti per ogni variabile che aggiungiamo le possibili combinazioni degli ingressi vengono raddoppiate ! (Continueremo ad usare il codice Gray) Avremo quindi per mappe a 4 variabili delle mappe quadrate i cui minterm saranno composti da 4 letterali.

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.

4.1 Multiplexer

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

4.1.1 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 Gray 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.

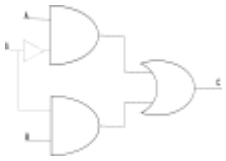


Figure 4.1: schema circuitale di un Multiplexer 2 a 1

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.

4.1.2 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_0$1 si occuperà di scremare tra A e B mentre $S_0$2 scremerà tra C e D.

4.2 Odd Function

E' una funzione che vale 1 SOLO se il numero di ingressi ad 1 è dispari.
Iniziamo sviluppando la tabella di Karnaugh.

a\ bc	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Purtroppo i term in questo caso coincidono con i minterm della tabella di karnaugh e sono : ab'c' , a'b'c , abc , a'bc.

Quindi la funzione logica dell' odd counter è :

f =ab'c'+a'b'c+abc+a'bc'

Possiamo ora procedere a raggruppare eventuali term sfruttando l'adiacenza logica.

f =b'(ac'+a'c)+b(ac+a'c') che possiamo ulteriormente compattare riconoscendo in ac'+a'c l'operatore booleano XOR e in ac+a'c l'operatore booleano XNOR.

4.3 Conta Uni

Lo scopo di questo circuito è dire in uscita quanti degli ingressi sono a 1. Nel nostro caso ci saranno 3 ingressi, quindi il numero in uscita potrà variare da 0 a 3 ossia da 00 a 11 in binario.

Chiamando gli ingressi a,b,c e le uscite s1 ed s2 , agiamo sulle uscite con due mappe di karnaugh distinte ma che in realtà sono parallele e sinergiche nel circuito.

Iniziamo con s1, che rappresenterà il bit meno significativo del risultato e perciò dovrà essere 1 nel caso in cui in ingresso ci siano 1 (01) o 3(11) "uni" in ingresso.

a \ bc	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Per s2 l'operazione sarà la stessa con la differenza che l'uscita sarà ad 1 nei casi in cui la quantità di "uni" in ingresso sia 2 (10) o 3 (11).

a\ bc	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Possiamo perciò ricavare le funzioni logiche per l'uscita s1 ed s2.

- S1
  Osservando la mappa di Karnaugh si può riconoscere nella funzione s1 la stessa forma che poco fa abbiamo chiamato odd function, infatti è proprio quella la funzione di questa uscita.

  Perciò f_{S1} =ab'c'+a'b'c+abc+a'bc'

- S2
  Qui invece richiede la ricerca da parte nostra di term all'interno della mappa, da cui otteniamo la seguente funzione : f_{s2} =bc+ac+ab .

4.4 Codifica e decodifica

4.4.1 Priority Encoder

Il priority encoder è un circuito la cui uscita corrisponderà all'indice dell'input con maggior precedenza ad 1. Quindi prima di tutto è necessario assegnare ad ogni ingresso un indice, ossia un valore che lo identifichi. Nel nostro caso avremmo tre ingressi : a,b,c e perciò una possibile corrispondenza per gli indici è l'uso dei numeri 1,2,3 rispettivamente. Per nostra scelta imponiamo che a(ossia 1) sia quello con priorità minore, mentre c(ossia 3) quello con priorità maggiore.

(Gli indici saranno smostrati in formato binario e perciò dato che abbiamo solo 3 ingressi basteranno 2 uscite perchè saranno da mostrare solo i valori 00, 01,10,11).

Procediamo quindi con la tabella della verità:

a(1)	b(2)	c(3)	s1	s2
0	0	0	0	0
0	0	1	1	1
0	1	1	1	1
1	1	1	1	1
1	0	1	1	1
0	1	0	1	0
1	1	0	1	0
1	0	0	0	1

Passiamo ora alle mappe di Karnaugh.

S2, che rispecchierà la cifra meno significativa del risultato, avrà la seguente tabella:

a\ bc	00	01	11	10
0	0	1	1	0
1	1	1	1	0

S1 invece che rispecchierà la cifra più significativa del risultato sarà:

a\ bc	00	01	11	10
0	0	1	1	1
1	0	1	1	1

Perciò le equazioni che rispecchiano le uscite sono :

f_{s1} =c+b

f_{s2} =c+ab'

E il circuito rappresentante il priority encoder è il seguente:

4.5 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
Quelli 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 Ottenere il diagramma temporale di un circuito

Prima di tutto è necessario identificare i nodi interni e assegnargli un nome identificativo.

Poi si procede identificando gli stimoli di ingresso e successivamente si valuta l'uscita di ogni nodo sulla base degli ingressi che ha ricevuto.

5.2 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**

5.2.1 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

Siccome il numero degli ingressi cresce con il numero delle cifre ben presto le dimensioni di input diventano ingestibili ecco che quindi si può ricorrere ad un metodo gerarchico, ossia un metodo iterativo che sfrutta un'operazione atomica e la replicaquante volte è necessario.

6.2 Somma

6.2.1 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 che 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 f_s =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 f_c =aANDb .
Il circuito dell'half adder è:

6.2.2 Full adder

L'half adder però non tiene conto di un'evetuale riporto proveniente dal calcolo fatto in precedenza. Infatti se vogliamo sfruttare un'architettura gerarchica bisogna tener conto che la somma atomica fatta per la cifra con un grado di significatività in meno rispetto a quella che stiamo sommando potrebbe aver generato un carry con la conseguente necessità di doverne tener conto ora che facciamo la somma.

Il problema si risolve aggiungendo un ingresso all'half adder per poter tenere conto di un0 eventuale carry non nullo.

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

La mappa di Karnaugh per il valore di S è :

c\ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

La mappa di Karnaugh per il valore di C è :

c\ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

COnfrontando il circuito con quelli che abbiamo già visto ci si accorge che è lo stesso circuito del conta uni !

6.2.3 Sommatore ripple carry

Ora quindi possiamo unire i circuiti dell'half adder e del full adder per ottenere un sommatore con un arbitrario numero di ingressi. Analogamente possiamo ottenere un sommatore con arbitrario numero di ingressi anche con tutti e soli full adder a patto però di mettere a 0 l'adder della cifra meno significativa così da renderlo un half adder a tutti gli effetti.

Il numero di uscite del ripple carry è pari al numero di ingressi +1 perchè deve anche mostrare un eventuale riporto dovuto all'ultimo sommatore.

Prestazioni

Il sommatore ripple carry ha dimensioni ridotte che aumentano linearmente con l'aggiunta di degli ingressi.

La catena dei carry forma un cammino molto lungo e ciò fa si che il sommatore sia lento con una lentezza proporzionale al numero di bit .

Ciò fa si che sussistano delle alee impossibili da eliminare che fanno cambiare il risultato finchè il carry non si è propagato fino all'ultimo full adder. Questo perchè gli adder calcolano la somma prima che un eventuale carry venga comunicato dagli adder che si occupano delle cifre meno significative e quindi in caso di carry non nullo dovrà rifare il calcolo per quella cifra.

6.2.4 Sommatore carry look-ahead

Permette di ottimizzare il calcolo del carry riducendo a 2 livelli questa operazione. Ciò a però complica la realizzazione.

Si comincia separando la somma dal carry.

6.3 Sottrazione

Molto siile alla somma però ora ci possono essere dei prestiti dalle cifre più significative. Inoltre il risultato può essere negativo.

Nel caso di risultati negativi, il calcolo si fa scambiando i numeri e poi mettendo un bit di segno.

Scambiare però i numeri è come scambiare due cavi, è un'operazione complicata !

Molto più semplice è l'uso di **notazione in complemento a 2**.

6.3.1 Notazione in complemento a 2

La notazione in complemento a 2 permette di rappresentare i numeri negativi in binario in un modo che a noi torna comodo per poterci fare operazioni algebriche. Solitamente con n bit a disposizione noi avremmo la possibilità di rappresentare 2^n numeri in binario, che sono i numeri da 0 (n zeri) fino a $2^n - 1$ (n uni).

Con la notazione in complemento a 2 invece un qualsiasi numero M è rappresentato come $2^n - M$.

- $M < 2^n$ Per esempio avendo 3 bit [*][*][*] e volendo rappresentare il numero 5 in binario si avrebbe [1][0][1]. In complemento a 2 questo è [1][1][1]-[1][0][1]=[0][1][0] ossia 2.
- $M > 2^n$ Per esempio avendo 3 bit [*][*][*] e volendo rappresentare il numero -3 in binario in complemento a 2 , [1][1][1]+[0][1][1]= [1][0][1][0] ossia 9.

Questo perchè mettendo in una tabella il complemento a 2 si ottiene il seguente risultato:

4	2	1	M	-4	2	1	M
0	0	0	0	0	0	0	0
1	1	1	7	1	1	1	-1
1	1	0	6	1	1	0	-2
1	0	1	5	1	0	1	-3
1	0	0	4	1	0	0	-4
0	1	1	3	0	1	1	3
0	1	0	2	0	1	0	2
0	0	1	1	0	0	1	1

6.4 Moltiplicazione

Volendo moltiplicare due numeri POSITIVI si può ancora una volta sfruttare la struttura gerarchica.

Utilizzando la rappresentazione binaria le moltiplicazioni tra due numeri possono essere costituite dalle seguenti combinazioni :

a	b	res
0	0	0
1	0	0
0	1	0
1	1	1

Quello sopra descritto è esattamente il comportamento della porta AND, per convincersene basta confrontare le due tabelle della verità.

Perciò il prodotto di due numeri ciascuno costituito da 1 bit ha come risultato l'operazione logica AND tra di essi.

Rendiamo ora il moltiplicatore multi-bit tramite struttura gerarchica:

Dati due numeri a e b rispettivamente con m e n bit ciascuno, il moltiplicatore a*b sarà una concatenazione in cui il bit meno significativo di b moltiplicherà tutto a e questo si fa mettendo m porte and in cui entrano una cifra di a e la meno significativa di b. Poi ciò produce un risultato, il cui bit meno significativo sarà il bit meno significativo del risultato finale mentre invece il resto del risultato è parziale e dovrà essere sommato al risultato della moltiplicazione tra le cifre di a e il bit secondo bit meno significativo. Iterando questo processo si ottiene il risultato cercato.

Attenzione che mentre la somma tra due numeri con n cifre può produrre al massimo un numero con n cifre+1 il prodotto tra due numeri di m ed n cifre può produrre un numero di cifre fino a 2*n, in quanto ogni moltiplicazione produce n cifre e per ogni risultato parziale ci sarò uno shift verso sinistra di 1 quindi alle n cifre si sommano n-1 cifre +1 per un eventuale carry out .

Il circuito del moltiplicatore multi-bit è il seguente :

Chapter 7

Linguaggi di descrizione dell'hardware

Una volta progettato un circuito e disegnato si potrebbe procedere a comprare le porte logiche ed assemblarlo ma la natura di test che quindi impone un alto tasso di variabilità del circuito durante la fase di progettazione ed affinamento e la crescente dimensione dei circuiti al giorno d'oggi non favoriscono questo approccio. Molto meglio è invece utilizzare un linguaggio di descrizione che permette di simulare il funzionamento delle porte e quindi dei circuiti.

7.1 Programmi richiesti: 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

Comandi frequenti

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
```

7.2 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.

7.2.1 Entità

La nozione di **entità** in VHDL corrisponde alla rappresentazione di un blocco **SENZA SPECIFICARNE** la logica interna ma solamente l'interfaccia con l'esterno.

Esso specifica :

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

Di seguito il codice per dichiarare un'entità di nome "name_entity" con n ingressi e m-n uscite.

```
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 entity nome_entity;
```

7.2.2 Architettura

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

```

architecture nome_architettura of nome_entita is
--dichiarazione di eventuali signal o buffer ausiliari
begin
--le equazioni e le operazioni di assegnamento che verranno scirtte
     in questa zona verranno eseguite tutte in contemporanea,
    indipendentemente dalla sequenza in cui compariranno


end architecture nome_architettura;
```

ATTENZIONE ! a differenza di linguaggi come il C, in VHDL l'operatore di assegnazione è '¡=' mentre quello di comparazione di uguaglianza è '='.

Per l'assegnazione si scrive a¡=b per indicare che il valore di b viene assegnato ad a.

7.2.3 Testbench

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

```
1
2  --dichiaro un entita senza interfaccia perche non ha necessita di
       porte di ingresso ne di uscita.
3  entity TestBench is
4  end entity TestBench;
5
6  --definisco ora la sua architettura
7
8  architecture test of TestBench is
9
10 --segnali interni di interconnessione
11 signal a,b,c : bit;
12
13 begin
14 --istanzio il modulo da testare
15 g1 : entitadatestare port map()
16
17 --definizione degli stimoli tramite equazioni in cui assegno ogni
       tot secondi un certo valore ai signal del mio testBench che a
       loro volta verranno assegnati alle porte dell entita da testare
       .
```

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.

7.2.4 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.

Questo si può fare alla **vista strutturale**.

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

```
1  component nomecomponent is
2
3  --descrizione dell' entita
4
5  end component;
```

Altrimenti si può fare direttamente all'interno dell'architettura che lo utilizzerà :

```
1
2  architecture nomeArchitetturaCheUtilizzeraComponent of nomeEntita
       is
3
4  begin
5
6    identificatoreIstanzaComponent : entity work.nomeComponent port
       map ();
7
8    --resto del codice
```

7.2.5 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.

Gli **identificatori** sono l'insieme di nomi utilizzati per entità, architetture e segnali.

VHDL è un linguaggio NON è 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 i signal sono visibili solo all'interno dell'architettura in cui sono dichiarati)
Inoltre a differenza di altri lignuaggi in VHDL l'ordine in cui sono dichiarate le equazioni nell'architettura non è importante perchè avvengono contemporaneamente.

7.2.6 Parametri formali ed effettivi

Le funzioni si richiamano nell'architettura tramite **port map** seguito da **notazione posizionale** e la loro instanziazione sarà contemporanea. 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**.

```
1 nome_istanza : nome_master port map(parametri effettivi)
```

portmap è un operatore che permette di mappare le porte dell'istnza dell'entità che si vuole utilizzare con segnali provenienti dall'entità contenitrice, così che possano lavorare insieme.

Si può anche optare per un'assegnazione di tipo esplicito e ciò aiuta a devitare

gli errori e ad indentificarli prima. Questo si fa con l'operatore "=¿" in cui a sx c'è la porta dell'entità che sto mappando e a destra un segnale dell'entità contenente.

```
1
2  architecture strutturale of Adder4 is
3
4  begin
5
6  FA0: entity work.full_adder port map (a=>A(0),b=>B(0),...);
7
8
9  --resto del codice
10
11 end architecture strutturale;
```

7.2.7 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 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 non essendo 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 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 : entita 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;
```

7.2.8 Tipi dei segnali

- **bit**
  Tipo di segnale che può assumere solamente i valori 0 ed 1.

- **bit_vector**
  Tipo di segnale che corrisponde ad un array di una certa dimensione. Si dischiara come

```
nomeSegnale : bit_vector(estremoSup downto estremoInf);
```

  oppure

```
nomeSegnale : bit_vector(estremoInf to estremoSup);
```

- **integer**
  Tipo di dato che corrisponde ad un intero a 32 bit. Poco conveniente da utilizzare.

- **time**

- **TIPI DEFINITI DALL'UTENTE**

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

- **TIPI COMPOSITI**

  Vanno dichiarati come prima cosa tra la dichiarazione dell'architettura che li userà e il begin, questo perchè così saranno utilizzabili da eventuali signal per esempio dichiarati subito dopo.

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 end record

  (Nei tipi struttura dove c'è scritto record dichiaro insieme di variabili che lo costituiscono)

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

La libreria standard 1164 della IEEE si può importare con il seguente codice:

```
--l'importazione della libreria va fatta in cima al codice
library ieee;
use ieee.std_logic.all;
```

L'introduzione di nuovi valori assumuibili dal tipo bit comporta una necessaria estensione della definizione degli operatori logici, ciò si chiama **overloading dell'operatore**.

Nella definizione dell'entità quindi ora una porta potrà essere invece di bit std_logic e std_logic_vector invece di bit_vector.

7.2.9 Espressioni condizionate

Sono operazioni che si basano sul principio del multiplexer e che vanno utilizzate nell'architettura di un'entità.
L'assegnazione può essere di due tipi:

- **Assegnazione Condizionata**
  Sono definite nella libreria ieee. Vanno implementate nell'architettura e sono tra di loro combinabili.

```
--architettura di un'entita
identificatore <= opzione1 when condizione else opzione2,
                  opzione2 ;
```

- **Assegnazione Selezionata**
  E' simile ad uno switch e utilizza un segnale per effettuare la decisione. La condizione può anche considerare più segnali così da essere molto flessibile.

```
with segnale select
identificatore <=opzione1 when condizione1,
                 opzione2 when condizione2,
                 opzione3 when condizione3 | condizione 4,
                 opzione4 when others;
```

La differenza principale tra di esse è che l'assegnazione condizionata può operare condizioni su segnali diversi mentre quella selezionata no. Inoltre da una priorità alle condizioni mentre quella selezionata no !

7.2.10 Stile Comportamentale

Permette di calcolare i valori di uscita tramite algoritmi. La peculiarità di questo stile è che a differenza di quanto visto fino ad ora le parti di codice sono eseguite in **ordine sequenziale**.

Processo:

- Compare all'interno di dell'architettura.
- Compare all'interno di una **sensitivity list**, ossia una lista di segnali che definiscono le condizioni di attivazioni del processo.
  Ogni volta che un parametro della lista varia il processo verrà eseguito.

```
architecture comportamentale of mio_modulo is
begin
label: process (sensitivity-list) is
  dichiarazioni di tipi,costanti,segnali,variabili;
  begin
   statement sequenziale;
   statement sequenziale;
   statement sequenziale;
  end process;
end architecture comportamentale;
```

Nello stile comportamentale possono essere usati "if then elsif else end if" all'interno di processi.

Variabili

Sono quantità definibili nei processi che vengono aggiornate immediatamente al momento dell'assegnazione mantenendo il loro valore tra un'attivazione e la successiva.

```

process (sensitivity-list) is

  variable nameVariable : tipodivariabile;

begin
  namevariabile:= valoredaassegnare

end process;
```

Cicli

```
1
2 process (sensitivity-list) is
3   --eventuali dichiarazioni
4 begin
5   for i in estremoSuperiore loop
6     --codice del ciclo
7   end loop;
8   --eventuali variabili da restituire andranno assegnate qui ad
      uscite o segnali cos' da non essere perse all'uscita dal ciclo.
9 end process;
```

- **Variabile i** viene dichiarata automaticamente.
- **Estremo superiore** è il valore a cui si smetterà di entrare nel ciclo quando i lo raggiungerà.

Chapter 8

Macchine a stati

8.1 Ciruiti sequenziali

Ci occupiamo ora di circuiti che possono **ricordare** la loro storia. QUesto sarà possibile grazie ad un nuovo ingresso che codificherà l'ultimo stato vissuto dal circuito.

Per semplicità conviene separare il circuito in due macro aree:

- Circuito Combinatorio
- Memoria

Come nella figura accanto
Un circuito sequenziale può essere di due tipi:

- **Sincrono**: Sono circuiti che aggiornano lo stato presente a cicli, detti **cicli di clock**
- **Asincrono**: Sono circuiti in cui lo stato presente continua ad aggiornarsi e ciò può essere compromettente per esempio nel caso di circuiti suscettibili a glitch.

Elementi di memoria

Gli elementi che verranno di seguito introdotti sono da considerare come black box, di cui verrà descritto solo il loro utilizzo mentre come implementano le porte logiche al loro interno per ottenere tali risultati verrà introdotto in altri corsi.

I circuiti che impiegano memorie sincrone hanno due stati:

- Clock basso e la rete combinatoria calcola.
- Clock alto e la rete combinatoria calcola con un nuovo stato presente.

Latch D

Il latch D ha due ingressi, di cui uno di clock e due ingressi Q e Q'.

Un latch durante il funzionamento del circuito vive due possibili stati che si alternano nel tempo:

- C=1 corrisponde a **latch trasparente** ossia stato in cui viene messo un nuovo valore in memoria.
- C=0 corrisponde a **latch in memoria** ossia stato in cui il valore in memoria viene conservato immutato.

Flip Flop Edge Triggered

Viene impiegato per risolvere i problemi di sfasamento tra frequenza di clock ed elaborazione del circuito.

Edge Triggered:

Significa che il flip flop diviene trasparente **SOLO** nella transizione (edge appunto) tra stato 0 ed 1 (o 1 e 0).

Preset & clear Permette di dare un valore noto al segnale iniziale del clock che altrimenti sarebbe incognito.

Stato

: è l'insieme dei valori conservati in memoria in un dato istante. E' implementato con un vettore di variabili booleane.

Equazioni di stato: