UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA MECCATRONICA

TESI DI LAUREA TRIENNALE

Simulazione di un moltiplicatore di Wallace a 8 bit descritto in VHDL

8 bit Wallace multiplier VHDL simulation

Relatore: Prof. SIMONE BUSO

Laureando: FEDERICO BARBOLAN

Matricola 561034-IMC

ANNO ACCADEMICO 2014-2015

Sommario

Nel presente elaborato, iniziando dalle basi fornite nel corso di “Teoria dei Circuiti Digitali”, viene affrontata la realizzazione e simulazione di un moltiplicatore a 8 bit fino alla sua ottimizzazione di “Wallace”.
Per l’esecuzione delle simulazioni verrà utilizzato l’ambiente di sviluppo “Xilinx”, iniziando dai sommatori definiti “elementari” per giungere successivamente a sommatori più “completi”. La realizzazione del moltiplicatore vera e propria viene ottenuta combinando opportunamente più sommatori tra di loro.
Ogni componente è stato implementato nel linguaggio VHDL, scegliendo di volta in volta i giusti componenti resi necessari a migliorare il moltiplicatore di Wallace.

Indice

Introduzione

In questo elaborato viene simulato il funzionamento di un moltiplicatore ad 8 bit e, facendo uso del programma di simulazione "Xilinx ISE Design Suite", è stato possibile implementare i vari componenti che lo compongono nel linguaggio VHDL.
Di volta in volta si sono studiati i componenti base, per poi passare a strutture più complesse ed infine realizzare il moltiplicatore vero e proprio facendo particolare attenzione all'implementazione in VHDL di ognuno di essi, scopo quindi della presente ricerca.
Inizialmente è stato preso in esame il sommatore più semplice, denominato Full-Adder, che costituisce il primo blocco fondamentale.
Successivamente sono stati sviluppati alcuni sommatori più complessi ad 8 bit. Il primo sommatore studiato è il sommatore Ripple-Carry, ma visti i limiti in termini di tempo di calcolo, viene preso in considerazione lo studio del sommatore Carry-Select che comporta un ottimo compromesso tra complessità e rapidità di calcolo.
Una volta realizzati i componenti fondamentali che costituiscono il moltiplicatore, si è proseguito nella realizzazione vera e propria del moltiplicatore ad 8 bit.
Il primo moltiplicatore analizzato è quello denominato Ripple-Carry ad Array che si mostra però essere lento nell'eseguire la moltiplicazione. Di seguito apportando alcune migliorie per ridurre i tempi di calcolo si è analizzato il moltiplicatore Carry-Save. Infine si è passati al moltiplicatore di "Wallace" ottenendo per tanto un notevole risparmio in termini di tempo di calcolo.

Il VHDL è un linguaggio per la descrizione dell'hardware (un *Hardware Description Language*), che può essere usato per la documentazione, la simulazione e la sintesi di sistemi digitali.
Il VHDL è stato introdotto come linguaggio standard per la documentazione di sistemi digitali complessi. Il linguaggio è nato quindi con lo scopo di fornire una descrizione non ambigua di un sistema digitale. Una caratteristica del linguaggio VHDL è la possibilità di simulare il sistema descritto, sia a livello funzionale sia portando in conto i ritardi del circuito.

1.1 Livelli di astrazione

Il VHDL è un linguaggio molto flessibile, riesce a fornire specifiche di circuiti digitali a diversi livelli di astrazione. Nei successivi paragrafi vedremo brevemente quali sono i livelli di astrazione del VHDL e le caratteristiche di ognuno di essi, in relazione alla struttura circuitale che sono in grado di realizzare.

1.1.1 Livello strutturale

Il livello strutturale è il più semplice livello di astrazione del linguaggio VHDL. In questo livello un sistema viene descritto mediante connessioni di opportuni componenti, in maniera analoga ad una rappresentazione mediante schema a blocchi. In una descrizione strutturale avremmo quindi dei componenti opportunamente collegati tra di loro attraverso appropriati segnali.

```
1
2  architecture Behavioral of RippleCarry_4 is
3
4  component FullAdder
5     port (A,B,Cin : IN std_logic;
6           S,Cout : OUT std_logic
7           );
8  end component;
9
10 signal C2,C1,C0 : std_logic;
11
12 begin
13
14    FA0 : FullAdder port map (A=>A(0), B=>B(0), Cin=>Cin, S=>S(0), Cout=>C0);
15    FA1 : FullAdder port map (A=>A(1), B=>B(1), Cin=>C0, S=>S(1), Cout=>C1);
16    FA2 : FullAdder port map (A=>A(2), B=>B(2), Cin=>C1, S=>S(2), Cout=>C2);
17    FA3 : FullAdder port map (A=>A(3), B=>B(3), Cin=>C2, S=>S(3), Cout=>Cout);
18
19 end Behavioral;
20
```

Figura 1.1 – *Sommatore Ripple-Carry implementato in VHDL utilizzando il livello strutturale.*

Prendiamo come esempio il listato di Figura 1.1, dove viene descritto un sommatore Ripple-Carry. Dalla riga 4 alla riga 8 viene dichiarato il componente che di seguito è utilizzato all'interno dell'architettura. Una volta istanziato un componente si deve utilizzare una sintassi come similarmente descritto dalla riga 14. La parola chiave "port map" introduce una lista che associa i port dell'entità con i segnali utilizzati nell'architettura. La lista può essere descritta in due diversi stili, posizionale o per corrispondenza per nome. Con il tipo posizionale, il primo segnale della lista corrisponde al primo port, il secondo segnale al secondo port e così via. Invece utilizzando una corrispondenza per nome ogni segnale viene collegato utilizzando l'operatore "=>" e l'ordine in cui vengono inseriti i segnali è ininfluente.

1.1.2 Livello data-flow

Il secondo livello di astrazione è il livello data-flow dove vengono descritte esplicitamente le trasformazioni che i dati subiscono durante la propagazione all'interno del circuito.

In particolare, il circuito è visto come due insiemi di elementi:

- Reti combinatorie: esprimono in maniera esplicita le trasformazioni dei dati mediante espressioni algebriche, espressioni aritmetiche e condizioni.
- Registri: sono deputati a memorizzare risultati intermedi di elaborazioni complesse.

È importante sottolineare che ogni operazione, intesa come elaborazione di dati, è assegnata esplicitamente ad un specifico ciclo di clock. Questa operazione prende il nome di scheduling ed è il progettista a farsene completamente a carico.

1.1.3 Livello algoritmico

Il livello algoritmico è il massimo livello di astrazione del VHDL. Con il livello algoritmico si è in grado di descrivere le funzionalità di un circuito mediante uno o più algoritmi. La struttura e le trasformazioni che subiscono i dati non sono esplicite, in particolare non sono esplicite le varie operazioni o elaborazioni assegnate ai cicli di clock. Sarà dunque lo strumento di sintesi ad effettuare lo scheduling sui cicli di clock in base ai vincoli imposti dal progettista.

1.1.4 Specifiche miste

Abbiamo visto i tre stili di descrizione: strutturale, data-flow, algoritmico. Il linguaggio VHDL è un linguaggio molto flessibile e consente di utilizzare tutti e tre gli stili all'interno di un'architettura.

1.2 Entity ed Architecture

Nel linguaggio VHDL per descrivere un sistema bisogna far riferimento a dei modelli. In particolare, bisogna definire l'interfaccia e il comportamento. L'interfaccia è quella parte di blocco dove vengono specificati i segnali di ingresso e di uscita di un sistema e consente di connettere il blocco stesso ad altri blocchi. Invece il comportamento descrive come gli ingressi vengono trasformati ed elaborati per produrre le uscite. L'interfaccia viene descritta mediante il costrutto "entity" mentre il comportamento viene descritto mediante il costrutto "architecture".
Per capire meglio questi concetti, prendiamo in esame il listato di Figura 1.2 che descrive un sommatore Full-Adder.

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4
5   entity FullAdder is
6       Port ( A : in  STD_LOGIC;
7              B : in  STD_LOGIC;
8              Cin : in  STD_LOGIC;
9              S : out  STD_LOGIC;
10             Cout : out  STD_LOGIC);
11  end FullAdder;
12
13  architecture Behavioral of FullAdder is
14
15  begin
16
17     S <= A xor B xor Cin;
18     Cout <= (A and B) or (B and Cin) or (A and Cin);
19
20  end Behavioral;
21
```

Figura 1.2 – *Listato di un Full-Adder completo di entità ed architettura.*

1.2.1 Dichiarazione dell'Entity

L'interfaccia di ogni modulo è descritta da una entity declaration. Tale costrutto specifica il nome e le porte del modulo. Nel listato di esempio della Figura 1.2 dalla riga 5 alla riga 11 viene dichiarato il costrutto entity. Il nome della entity, FullAdder, è indicato nella riga 5, successivamente vengono dichiarate le porte di ingresso e di uscita del sistema. Nel nostro caso abbiamo tre terminali di ingresso, denominati A, B, C_{in} e due terminali di uscita, denominati S e C_{out}. Tutti i terminali sono di tipo *STD_LOGIC* (vedremo nei paragrafi successivi i tipi utilizzabili nel linguaggio VHDL).
Una dichiarazione di entità è analoga ad un simbolo in uno schema a blocchi in cui si identificano il nome dell'elemento ed i punti di collegamento con altri elementi dello schema a blocchi. La Figura 1.3 mostra un simbolo schematico corrispondente all'entità descritta nel listato di Figura 1.2.

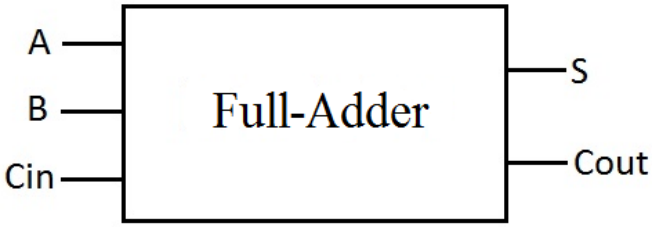

Figura 1.3 – *Schema concettuale dell'entità Full-Adder.*

1.2.2 Dichiarazione dell'Architecture

Abbiamo visto che una entity declaration definisce l'interfaccia di un modulo, ma non dice la funzionalità svolta dal modulo stesso. La funzionalità di un modulo è descritta in VHDL mediante una architecture declaration. Nel listato di Figura 1.2 dalla riga 13 alla riga 20 viene dichiarato il costrutto architecture. Il nome dell'architettura è Behavioral ed è riferito all'entità FullAdder e successivamente tra le parole chiave *begin* ed e*nd* viene dichiarata la funzionalità che il modulo deve svolgere. In questo semplice esempio i segnali di ingresso A, B, C_{in}, come richiesto ed evidenziato in precedenza, sono soltanto letti mentre i segnali di uscita S e C_{out} sono soltanto scritti.
La prima cosa che risulta evidente è che l'architettura si riferisce ad una sola entity, ma non è vero il contrario. Infatti è possibile specificare più architetture alternative e successivamente selezionarne una prima di procedere con la sintesi o la simulazione.
All'interno di un'architettura si possono specificare quattro tipi di dichiarazioni: costanti, segnali, tipi, componenti. Senza entrare nei specifici dettagli delle varie tipologie di dichiarazioni, si tenga presente che le medesime dichiarazioni all'interno dell'architettura hanno visibilità soltanto per quell'architettura.

1.3 Tipi

Il VHDL dispone di un numero elevato di tipi, tuttavia solo pochi di essi risultano utilizzabili dagli strumenti di sintesi automatica. Nei paragrafi seguenti vengono elencati i tipi base del linguaggio VHDL e sono introdotti alcuni concetti di base.

1.3.1 Il tipo bit

Il tipo *bit* è il più semplice tipo disponibile in VHDL. Tale tipo rappresenta il valore binario e può assumere solo due valori logici: 0 ed 1. Si noti che le due costanti 0 ed 1 devono essere racchiuse da apici ('0' ed '1') per distinguerli dai valori numerici 0 ed 1. Per il tipo *bit* si possono usare soltanto gli operatori di assegnazione, operatori di confronto ed operatori logici.

In Figura 1.4 viene mostrato un esempio degli operatori consentiti per il tipo bit.

```
1
2  Y <= A and B;
3  X <= '0';
4  Z <= B and (A not C);
5
```

Figura 1.4 – *Esempio di operatori utilizzabili per il tipo bit.*

Spesso è utile raggruppare i bit in array, utilizzando quindi un nome comune. Il linguaggio VHDL dispone a tale scopo del tipo *BIT_VECTOR.* In sostanza si tratta di un insieme di segnali contraddistinto da un nome e da un indice. Analizziamo la sintassi del tipo *BIT_VECTOR.*

```
1
2  Nome_Segnale : BIT_VECTOR ( indice1 {to|downto} indice2 );
3
```

Figura 1.5 – *Sintassi generica per inizializzare un array di tipo bit in VHDL.*

Il vettore di bit viene costruito tramite la parola chiave *BIT_VECTOR* (Figura 1.5), deve avere un nome che li contraddistingue. Inoltre il vettore ha un ordinamento che è determinato dalla parola chiave *to* o *downto*.
In particolare, utilizzando la formula *indice1 to indice2* il bit più significativo è quello con *indice1*, mentre nella forma *downto* il bit più significativo è quello in posizione *indice2.*

1.3.2 Il tipo integer

Il tipo *integer* rappresenta valori interi a 32 bit e può essere utilizzato per la sintesi. Questo tipo presenta due problemi. Il primo è come interpretare i 32 bit dal punto di vista del segno, come default i valori vengono rappresentati senza segno. Il secondo problema riguarda lo spreco di risorse, infatti lo strumento di sintesi considera sempre 32 bit ogni volta che un segnale intero viene utilizzato.

1.3.3 I tipi IEEE

In alcuni casi è necessario definire valori di un segnale diversi da 0 ed 1. Il VHDL non dispone di tale tipo, tuttavia esiste una libreria standard, la libreria IEEE, che definisce alcuni tipi aggiuntivi. I tipi in esame utilizzano un sistema a 9 valori, descritto in seguito:

’0’	Valore logico 0.
’1’	Valore logico 1.
’Z’	Alta impedenza.
’X’	Indeterminato. Può essere 0 o 1.
’U’	Indefinito. Il valore non è mai stato assegnato.
’W’	Segnale debole. Non è possibile interpretarlo come 0 o 1.
’L’	Segnale debole. Interpretabile come 0.
’H’	Segnale debole. Interpretabile come 1.
’-’	Don’t care.

Ogni libreria VHDL è suddivisa in package, ognuno dei quali definisce oggetti che poi vengono utilizzati nella progettazione. In particolare il package *STD_LOGIC_1164* definisce due tipi risolti *STD_LOGIC* ed *STD_ULOGIC* ed i corrispettivi vettoriali *STD_LOGIC_VECTOR* ed *STD_ULOGIC_VECTOR.*
Di particolare interesse sono i tipi *STD_LOGIC* ed *STD_LOGIC_VECTOR* in quanto sono tipi risolti.

Per chiarire meglio il concetto prendiamo l'esempio di Figura 1.6 dove su una linea abbiamo due valori logici diversi.

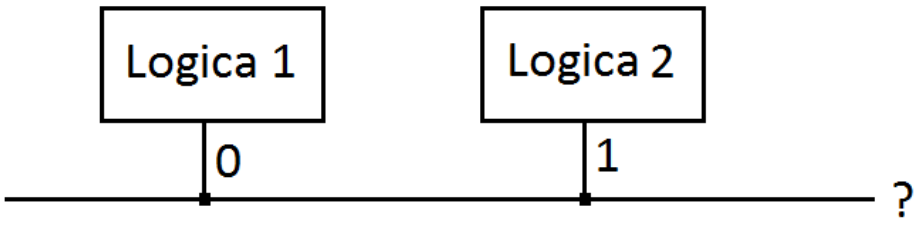

Figura 1.6 – *Valore assunto dalla linea utilizzando i tipi risolti.*

I tipi risolti permettono di definire il valore della linea mediante un insieme di regole associate alla logica a nove valori descritta in precedenza.
Inoltre, va sottolineato che il linguaggio VHDL proibisce la connessione di più segnali, a meno che per il tipo di segnali non siano definite una o più funzioni di risoluzione.

In elettronica digitale poter sommare parole di N bit è un operazione alquanto fondamentale. Occorrono quindi dei circuiti in grado di poter eseguire questa operazione in modo efficiente.

Volendo realizzare un sommatore di due parole ad N bit, la prima soluzione che viene in mente è quella di utilizzare sottoblocchi fondamentali. Il primo sottoblocco risulta essere quindi un sommatore che sia in grado di sommare 3 bit (due bit in ingresso ed un eventuale riporto). Successivamente questi sottoblocchi devono essere collegati in maniera opportuna: il sottoblocco i-esimo ha in ingresso il bit i-esimo della prima parola, il bit i-esimo della seconda parola ed il carry generato dal sottoblocco (i-1) -esimo. (Si assume che il blocco i=0 sia atto a sommare i LSB della parola in ingresso).
Il sottoblocco con posizione i=0, può essere sostituito con un blocco senza riporto in ingresso se viene considerato il primo blocco di una catena, tuttavia può esserne munito se si tratta di un "macro-sottoblocco" utilizzabile in una cascata più lunga. Analogamente il bit di riporto alla posizione N può essere utilizzato come parte del risultato (se il risultato finale è N+1 bit), oppure può rappresentare l'overflow (se il risultato prevede un massimo di N bit) oppure può essere propagato ad altri sommatori in cascata.

2.1 Sommatore Full-Adder

Il Full-Adder è il primo "sottoblocco" denominato in precedenza, ed è in grado di sommare tre bit. Viene detto anche sommatore completo in quanto può sommare due bit (A e B le due parole in ingresso) più un bit di riporto, chiamato C_{in}. È caratterizzato quindi da tre ingressi e due uscite. La tabella della verità risulta essere la seguente:

Tabella 1.1 – *Tabella della verità di un Full-Adder.*

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

Formula di base:

$$A + B + C_{in} = S + C_{out} \tag{1}$$

Utilizzando le nozioni di base dell'algebra booleana possiamo scrivere il risultato dell'uscita come:

$$S = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + ABC_{in} \tag{2}$$

$$S = \overline{A}(\overline{B}C_{in} + B\overline{C_{in}}) + A(\overline{B}\,\overline{C_{in}} + BC_{in}) \tag{3}$$

$$S = \overline{A}(B \oplus C_{in}) + A(\overline{B \oplus C_{in}}) \tag{4}$$

Se chiamiamo $X = B \oplus C_{in}$ (5)

$$S = \overline{A}X + A\overline{X} \tag{6}$$

$$S = A \oplus (B \oplus C_{in}) \tag{7}$$

Per il calcolo del riporto possiamo scrivere:

$$C_{out} = \overline{A}BC_{in} + A\overline{B}C_{in} + AB\overline{C_{in}} + ABC_{in} \quad (8)$$

Usando l'identità $X + X = X$ dove $X = ABC$ possiamo scrivere: (9)

$$C_{out} = BC_{in}(\overline{A} + A) + AC_{in}(\overline{B} + B) + AB(\overline{C_{in}} + C_{in}) \quad (10)$$

$$C_{out} = BC_{in} + AC_{in} + AB \quad (11)$$

La realizzazione in VHDL delle equazioni (7) e (11) sono mostrate in Figura 1.2 del capitolo precedente, mentre in Figura 2.1 viene mostrato un esempio di simulazione.

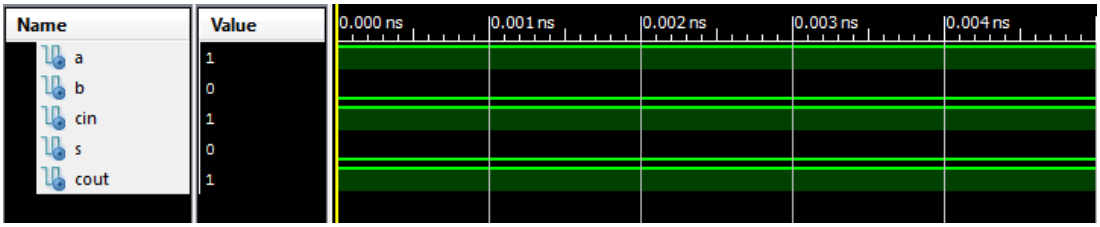

Figura 2.1 – *Esempio di simulazione del Full-Adder.*

2.2 Sommatore Ripple-Carry a 4 bit

Proseguendo con la terminologia adottata in precedenza costruiamo un "macro-sottoblocco" in grado di sommare due parole di 4 bit. Il principio di funzionamento del sommatore Ripple-Carry si basa sulla propagazione del riporto, ovvero il riporto dello stadio i-esimo sarà l'ingresso dello stadio (i+1)-esimo. Serviranno quindi 4 Full-Adders, ciascuno dei quali atto a sommare il bit i-esimo della parola. Il Full-Adder più significativo renderà disponibile il suo riporto, mentre il Full-Adder meno significativo potrà essere sostituito con un sommatore senza riporto. Nel nostro caso utilizziamo il riporto anche per il sottoblocco meno significativo, così da poterlo concatenare in una cascata di Ripple-Carry per creare sommatori a più bit.
Lo schema elettrico di principio del sommatore Ripple-Carry a 4 bit è riportato in Figura 2.2:

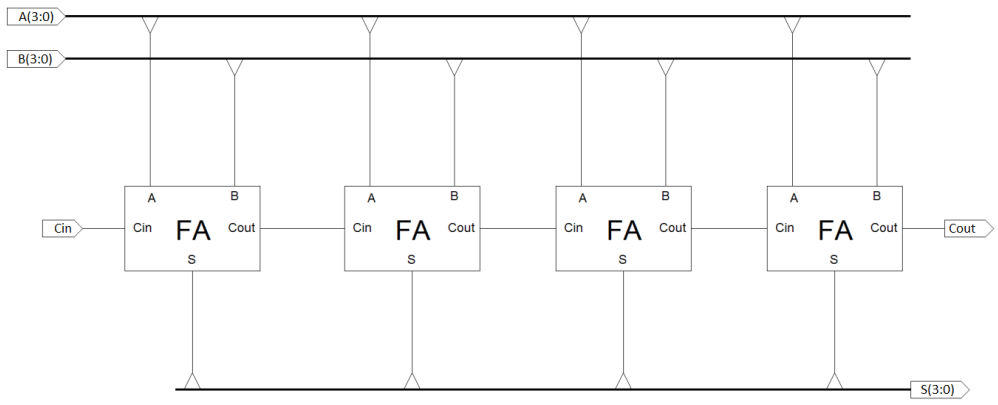

Figura 2.2 – *Schema elettrico di principio di un Ripple-Carry a 4 bit (http://www2.units.it/, "I Circuiti sommatori").*

Dallo schema di principio è possibile implementare in VHDL il sommatore Ripple-Carry a 4 bit come mostrato in Figura 2.3.
Si può notare come sia facile trascrivere nel linguaggio VHDL lo schema a blocchi precedentemente illustrato. Viene dichiarato il componente Full-Adder all'interno dell'architettura e successivamente, tramite il comando "port map" si creano le connessioni dei Full-Adder tra di loro.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity RippleCarry_4 is
6      Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
7             B : in  STD_LOGIC_VECTOR (3 downto 0);
8             Cin : in  STD_LOGIC;
9             S : out  STD_LOGIC_VECTOR (3 downto 0);
10            Cout : out  STD_LOGIC
11            );
12 end RippleCarry_4;
13
14 architecture Behavioral of RippleCarry_4 is
15
16 --dichiarazione dei componenti utilizzati per realizzare il sommatore Ripple Carry a 4 bit
17 component FullAdder
18    port (A,B,Cin : IN std_logic;
19          S,Cout : OUT std_logic
20          );
21 end component;
22
23 --dichiarazione dei segnali utilizzati per collegare internamente i componenti tra loro;
24 signal C2,C1,C0 : std_logic;
25
26 begin
27
28    FA0 : FullAdder port map (A=>A(0), B=>B(0), Cin=>Cin, S=>S(0), Cout=>C0);
29    FA1 : FullAdder port map (A=>A(1), B=>B(1), Cin=>C0, S=>S(1), Cout=>C1);
30    FA2 : FullAdder port map (A=>A(2), B=>B(2), Cin=>C1, S=>S(2), Cout=>C2);
31    FA3 : FullAdder port map (A=>A(3), B=>B(3), Cin=>C2, S=>S(3), Cout=>Cout);
32
33 end Behavioral;
34
```

Figura 2.3 – *Implementazione in VHDL del Ripple-Carry a 4 bit.*

2.3 Sommatore Ripple-Carry a 8 bit

Un sommatore Ripple-Carry a 8 bit è facilmente realizzabile collegando in cascata due sommatori Ripple-Carry a 4 bit. In Figura 2.4 viene mostrato lo schema di principio del moltiplicatore:

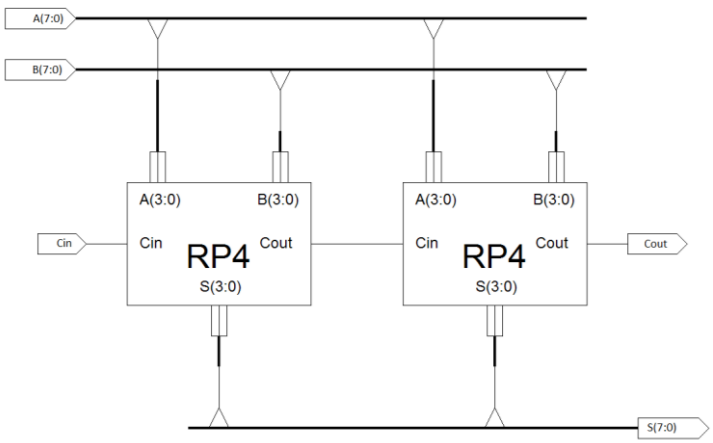

Figura 2.4 – *Schema elettrico di principio di un Ripple-Carry a 8 bit (http://www2.units.it/, "I Circuiti sommatori").*

Si noti che il primo stadio è munito di riporto in ingresso, così da poter essere collegato con altri "macro-sottoblocchi" per realizzare moltiplicatori a più bit. Come in precedenza, dallo schema di principio si passa alla realizzazione VHDL, mostrata in Figura 2.5:

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity RippleCarry_8 is
6      Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
7             B : in  STD_LOGIC_VECTOR (7 downto 0);
8             Cin : in  STD_LOGIC;
9             S : out  STD_LOGIC_VECTOR (7 downto 0);
10            Cout : out  STD_LOGIC
11           );
12 end RippleCarry_8;
13
14 architecture Behavioral of RippleCarry_8 is
15
16 --dichiarazione dei componenti utilizzati per realizzare il sommatore Ripple Carry a 8 bit
17 component RippleCarry_4
18    port (A,B : in STD_LOGIC_VECTOR (3 downto 0);
19          Cin : in STD_LOGIC;
20          S : out STD_LOGIC_VECTOR (3 downto 0);
21          Cout : out STD_LOGIC
22          );
23 end component;
24
25 --dichiarazione dei segnali utilizzati per collegare i componenti tra loro;
26 signal C : STD_LOGIC;
27 signal A1, A0, B1, B0 : STD_LOGIC_VECTOR (3 downto 0);
28
29 begin
30
31    A1 <= A (7 downto 4);
32    A0 <= A (3 downto 0);
33    B1 <= B (7 downto 4);
34    B0 <= B (3 downto 0);
35
36    RC4_0 : RippleCarry_4 port map (A=>A0, B=>B0, Cin=>Cin, S=>S(3 downto 0), Cout=>C);
37    RC4_1 : RippleCarry_4 port map (A=>A1, B=>B1, Cin=>C, S=>S(7 downto 4), Cout=>Cout);
38
39 end Behavioral;
40
```

Figura 2.5 – *Implementazione in VHDL del Ripple-Carry a 8 bit.*

Come per il sommatore a 4 bit, risulta molto semplice implementare in VHDL un sommatore ad 8 bit. Viene inizializzato il componente Ripple-Carry a 4 bit e, successivamente, due istanze del componente vengono collegate tramite il comando "port map" avvalendosi di segnali di supporto.

2.4 Sommatore Carry-Look-Ahead

Il sommatore Ripple-Carry appena visto potrebbe essere esteso per un numero qualsiasi di bit, ma presenta un problema: la lentezza. Infatti, all'aumentare del numero di bit della parola da sommare questo sommatore diventa via via sempre più lento nell'eseguire il calcolo. Questo è dovuto al fatto che per calcolare il bit più significativo, il Full-Adder alla posizione i=7 deve aspettare il resto del Full-Adder precedente (i=6), che a sua volta deve aspettare il resto dello stadio con posizione i=5 e così via per tutta la catena di Full-Adder. Si determina che la somma finale risulta attendibile quando tutti gli stadi hanno calcolato somma e riporto. Per ovviare il problema nasce il sommatore Carry-Look-Ahead, il quale riesce a limitare la lentezza facendo in modo che il resto venga calcolato da una logica combinatoria a parte. Questo tipo di sommatore è in grado di migliorare notevolmente la velocità e, inoltre, riesce ad essere indipendente dalla lunghezza delle parole da sommare. Tuttavia il circuito che prevede i risultati di carry risulta essere estremamente complesso. In questo trattato non viene implementato nel linguaggio VHDL il seguente sommatore proprio per la sua complessità.

Solo a scopo di esempio in Figura 2.6 viene mostrato un segmento di circuito per il solo calcolo del riporto del bit più significativo.

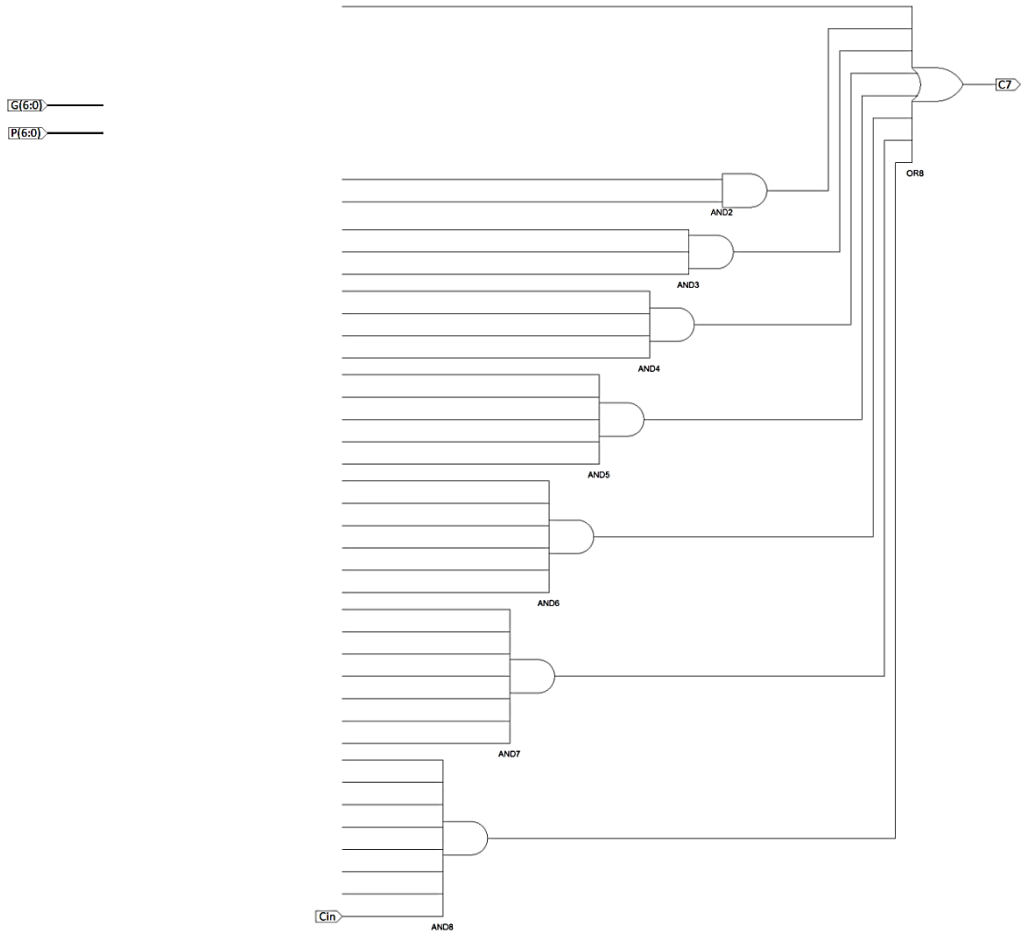

Figura 2.6 – *Porzione di logica combinatoria per il calcolo del bit più significativo di un sommatore Carry-Look-Ahead (http://www2.units.it/, "I Circuiti sommatori").*

2.5 Sommatore Carry-Select a 8 bit

Abbiamo visto fino ad ora come il sommatore Ripple-Carry sia abbastanza semplice da realizzare ma presenta il grosso problema della lentezza, mentre il sommatore Carry-Look-Ahead sia estremamente veloce, ma molto complesso da realizzare. Per ovviare a tali problemi è necessario analizzare il sommatore Carry-Select. Questo sommatore spezza la parola da sommare di N bit in parole più piccole M=N/2 e fa uso complessivamente di tre sommatori Ripple-Carry a M bits. Il risultato è che ogni sotto-sommatore risulta più semplice, più veloce nell'eseguire l'operazione di somma in quanto la parola risulta più breve; come svantaggio però si dovrà utilizzare un sommatore in più rispetto a quelli necessari.
Vediamo in seguito come realizzare il sommatore Carry-Select:

- Un Ripple-Carry serve per sommare i LSB, producendo quindi in uscita i bit meno significativi e un riporto C che servirà per lo stadio successivo;
- Un Ripple-Carry serve per sommare, in contemporanea al precedente, i MSB in caso che C valga 0;
- Un Ripple-Carry serve per sommare, in contemporanea al precedente, i MSB in caso che C valga 1;
- Un "multiplexer vettoriale" in grado di scegliere i MSB e il riporto finale in base al valore di C prodotto dal primo Ripple-Carry.

Il multiplexer vettoriale

I comuni multiplexer sono dispositivi in grado di selezionare un ingresso tra i vari ingressi possibili e, tramite gli ingressi di selezione, trasferirlo in uscita. Il nostro problema, invece, è poter portare in uscita cinque delle dieci linee disponibili in ingresso tramite un solo ingresso di selezione. Abbiamo quindi bisogno di un dispositivo in grado di fornire un bus di uscita formato da 5 linee (4 MSB prodotti dai Ripple-Carry precedenti più un bit di riporto) con due bus in ingresso da 5 linee ciascuno. Vediamo uno schema elettrico di tale dispositivo, raffigurato in Figura 2.7.

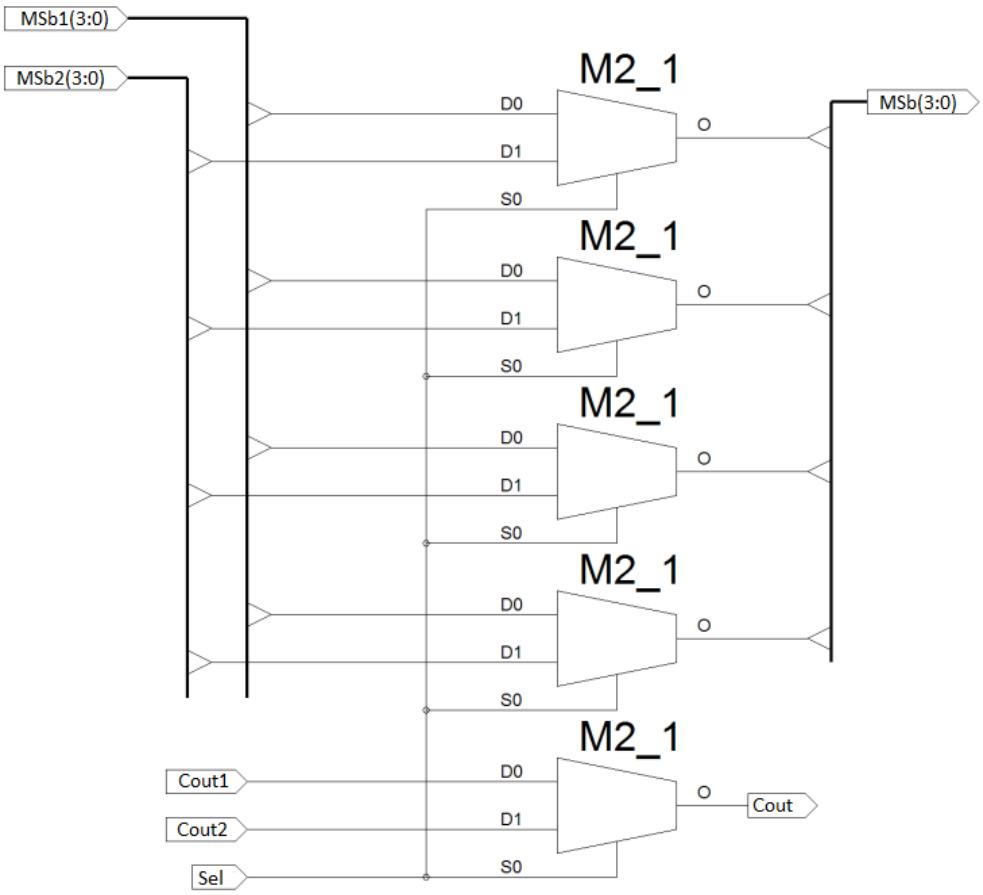

Figura 2.7 – *Schema elettrico di principio di un multiplexer vettoriale (http://www2.units.it/, "I Circuiti sommatori").*

Sommatori

L'implementazione in VHDL risulta essere la seguente: (Figura 2.8)

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4
5
6   entity VMux is
7       Port ( MSb1 : in  STD_LOGIC_VECTOR (3 downto 0);
8              MSb2 : in  STD_LOGIC_VECTOR (3 downto 0);
9              Cout1 : in  STD_LOGIC;
10             Cout2 : in  STD_LOGIC;
11             Sel : in  STD_LOGIC;
12             MSb : out  STD_LOGIC_VECTOR (3 downto 0);
13             Cout : out STD_LOGIC
14            );
15  end VMux;
16
17  architecture Behavioral of VMux is
18
19  component Mux is
20      Port ( A : in  STD_LOGIC;
21             B : in  STD_LOGIC;
22             Y : out  STD_LOGIC;
23             Sel : in  STD_LOGIC
24            );
25  end component;
26
27  begin
28
29     M3 : Mux port map (A=>MSb1(3), B=>MSb2(3), Y=>MSb(3), Sel=>Sel);
30     M2 : Mux port map (A=>MSb1(2), B=>MSb2(2), Y=>MSb(2), Sel=>Sel);
31     M1 : Mux port map (A=>MSb1(1), B=>MSb2(1), Y=>MSb(1), Sel=>Sel);
32     M0 : Mux port map (A=>MSb1(0), B=>MSb2(0), Y=>MSb(0), Sel=>Sel);
33     Mc : Mux port map (A=>Cout1, B=>Cout2, Y=>Cout, Sel=>Sel);
34
35  end Behavioral;
36
```

Figura 2.8 – *Implementazione in VHDL del multiplexer vettoriale.*

Si noti che per realizzare il multiplexer vettoriale si è fatto uso di 5 multiplexer a due ingressi che, collegati secondo lo schema sopra illustrato, riescono a fornire in uscita i 4 MSB più il carry.
Dopo aver definito il multiplexer vettoriale si può implementare il sommatore Carry-Select secondo lo schema di principio seguente: (Figura 2.9)

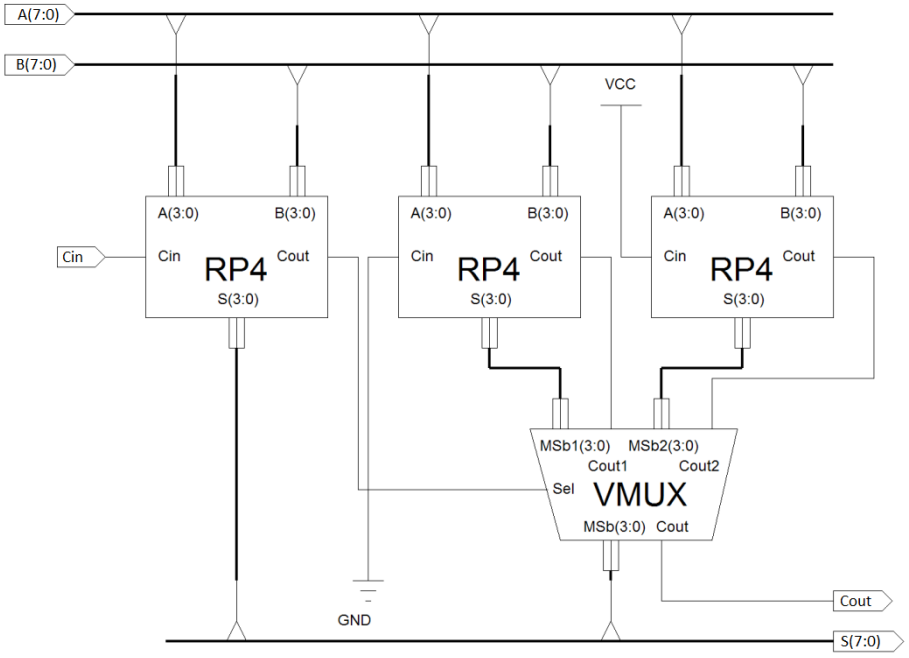

Figura 2.9 – *Schema elettrico di principio di un Carry-Select a 8 bit (http://www2.units.it/, "I Circuiti sommatori").*

Il sommatore Carry-Select a 8 bit implementato in VHDL è mostrato in Figura 2.10:

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity CarrySelect_8 is
6      Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
7             B : in  STD_LOGIC_VECTOR (7 downto 0);
8             Cin : in  STD_LOGIC;
9             S : out  STD_LOGIC_VECTOR (7 downto 0);
10            Cout : out  STD_LOGIC);
11 end CarrySelect_8;
12
13 architecture Behavioral of CarrySelect_8 is
14
15 component RippleCarry_4
16    port (A,B : in STD_LOGIC_VECTOR (3 downto 0);
17          Cin : in STD_LOGIC;
18          S : out STD_LOGIC_VECTOR (3 downto 0);
19          Cout : out STD_LOGIC
20          );
21 end component;
22
23 component VMux is
24     Port ( MSb1 : in  STD_LOGIC_VECTOR (3 downto 0);
25            MSb2 : in  STD_LOGIC_VECTOR (3 downto 0);
26            Cout1 : in  STD_LOGIC;
27            Cout2 : in  STD_LOGIC;
28            Sel : in  STD_LOGIC;
29            MSb : out  STD_LOGIC_VECTOR (3 downto 0);
30            Cout : out STD_LOGIC
31           );
32 end component;
33
34 signal Sel_v, Cout_1, Cout_2 : STD_LOGIC;
35 signal bit1, bit2 : STD_LOGIC_VECTOR (3 downto 0);
36
37 begin
38
39    RC4_0 : RippleCarry_4
40            port map (A=>A(3 downto 0), B=>B(3 downto 0), Cin=>Cin, S=>S(3 downto 0), Cout=>Sel_v);
41    RC4_1 : RippleCarry_4
42            port map (A=>A(7 downto 4), B=>B(7 downto 4), Cin=>'0', S=>bit1, Cout=>Cout_1);
43    RC4_2 : RippleCarry_4
44            port map (A=>A(7 downto 4), B=>B(7 downto 4), Cin=>'1', S=>bit2, Cout=>Cout_2);
45    VM : VMux
46         port map (MSb1=>bit1, MSb2=>bit2, Cout1=>Cout_1, Cout2=>Cout_2, Sel=>Sel_v, MSb=>S(7 downto 4), Cout=>Cout);
47
48 end Behavioral;
49
```

Figura 2.10 – *Implementazione in VHDL del Carry-Select a 8 bit.*

Per implementare il sommatore in VHDL si inizializzano il componente 'Vmux' (multiplexer vettoriale) e il sommatore Ripple-Carry a quattro bit. Premesso ciò, viene implementata l'architettura e, aiutandosi con i segnali di supporto, si riescono a collegare i vari sottoblocchi, come richiesto dallo schema. In particolare il primo stadio (RC4_0) porta in uscita i 4 bit meno significativi e rende disponibile il resto (chiamato 'Sel_v'), in contemporanea i due sommatori RC4_1 e RC4_2 calcolano i MSB ed il resto. Infine il Multiplexer vettoriale "sceglie" quale Ripple-Carry portare in uscita tramite il segnale 'Sel_v'.

2.6 Sommatore Carry-Select a 16 bit

Nel paragrafo precedente abbiamo realizzato un sommatore Carry-Select a 8 bit, costruendo quindi un "macro-sottoblocco". Ora per creare un sommatore a più bit, precisamente in questo caso a multipli di 8, è sufficiente collegare in cascata più "macro-sottoblocchi".

L'implementazione in VHDL di tale sommatore risulta abbastanza semplice come mostrato in Figura 2.11:

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity CarrySelect_16 is
6      Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
7             B : in  STD_LOGIC_VECTOR (15 downto 0);
8             Cin : in  STD_LOGIC;
9             S : out  STD_LOGIC_VECTOR (15 downto 0);
10            Cout : out  STD_LOGIC
11           );
12 end CarrySelect_16;
13
14 architecture Behavioral of CarrySelect_16 is
15 --dichiarazione dei componenti utilizzati per realizzare il sommatore Ripple Carry a 8 bit
16
17 component CarrySelect_8
18    port (A,B : in STD_LOGIC_VECTOR (7 downto 0);
19          Cin : in STD_LOGIC;
20          S : out STD_LOGIC_VECTOR (7 downto 0);
21          Cout : out STD_LOGIC
22          );
23 end component;
24 --dichiarazione dei segnali utilizzati per collegare i componenti tra loro;
25
26 signal C : STD_LOGIC;
27
28 begin
29
30    RC8_0 : CarrySelect_8 port map (A=>A(7 downto 0), B=>B(7 downto 0), Cin=>Cin, S=>S(7 downto 0), Cout=>C);
31    RC8_1 : CarrySelect_8 port map (A=>A(15 downto 8), B=>B(15 downto 8), Cin=>C, S=>S(15 downto 8), Cout=>Cout);
32
33 end Behavioral;
34
```

Figura 2.11 – *Implementazione in VHDL del Carry-Select a 16 bit.*

Per eseguire una moltiplicazione tra due parole di N bit ciascuna, si utilizza l'algoritmo più semplice, nello specifico quello elementare (ovvero la moltiplicazione classica, usata in base 10). In tabella 3.1 viene mostrata la matrice dei prodotti parziali e la somma finale. In questo caso partiamo da un esempio semplice di moltiplicazione di due parole da 4 bit. Nell'effettuare le somme si possono generare dei riporti che bisogna opportunamente portare in conto. Infine, date due parole da N bit, la somma finale sarà rappresentata su 2xN bit.
Ci sono due tipi di moltiplicatori hardware: moltiplicatore seriale e moltiplicatore parallelo. Successivamente vedremo in dettaglio e l'implementazione in VHDL di entrambi i tipi di moltiplicatore.

Tabella 3.1 – *Matrice dei prodotti parziali.*

				X_3	X_2	X_1	X_0
				Y_3	Y_2	Y_1	Y_0
				Y_0X_3	Y_0X_2	Y_0X_1	Y_0X_0
			Y_1X_3	Y_1X_2	Y_1X_1	Y_1X_0	
		Y_2X_3	Y_2X_2	Y_2X_1	Y_2X_0		
	Y_3X_3	Y_3X_2	Y_3X_1	Y_3X_0			
S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

3.1 Moltiplicatore Ripple-Carry ad Array

La prima architettura hardware, ovvero moltiplicatore seriale, è mostrata in Figura 3.1 e viene riportata una struttura a 4 bit.
Come si può notare dallo schema di principio abbiamo una struttura regolare e ogni blocco è opportunamente interconnesso. Ogni cella calcola il bit di somma S, lo stesso viene inviato alla cella inferiore, ed un eventuale riporto C_{out}, che può essere inviato alla cella a sinistra, essendo di peso superiore.

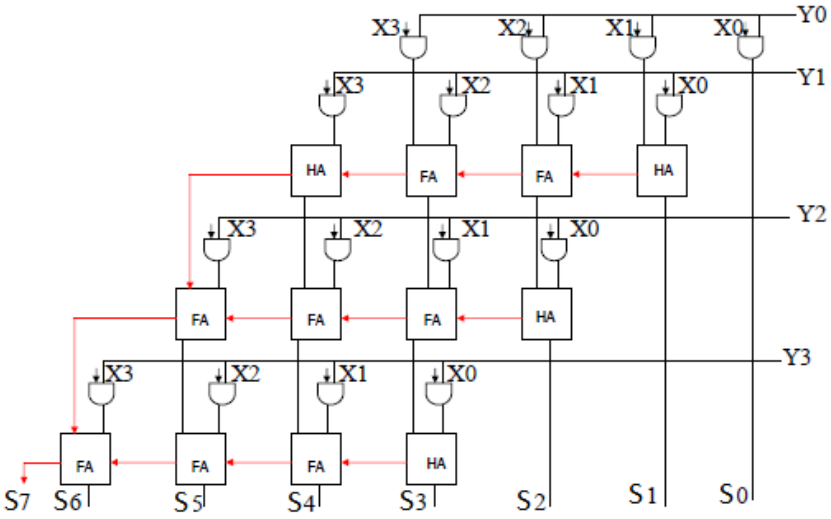

Figura 3.1 – *Schema elettrico di principio di un moltiplicatore Ripple-Carry ad Array a 4 bit (http://www.unibo.it/, "Elettronica dei Sistemi Digitali LA").*

Moltiplicatori a 8 bit

Come mostrato in Figura 3.1, prima si creano i blocchi elementari in grado di svolgere il calcolo dei prodotti parziali, questa operazione viene svolta da semplici porte AND. Successivamente, utilizzando dei sommatori precedentemente studiati e collegandoli in cascata tra di loro, si realizza la somma di coppie di prodotti parziali per poi ottenere la somma totale per colonne.
Ora, per implementare un sommatore Ripple-Carry ad Array ad 8 bit in VHDL, possiamo utilizzare lo schema di principio mostrato in Figura 3.2:

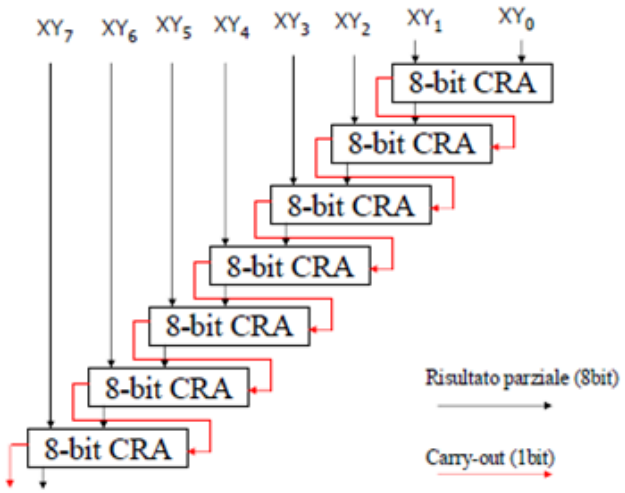

Figura 3.2 – *Schema di principio di un moltiplicatore Ripple-Carry ad Array a 8 bit (http://www.unibo.it/, "Elettronica dei Sistemi Digitali LA").*

L'implementazione nel linguaggio VHDL risulta essere la seguente: (Figura 3.3)

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity Moltiplicatore_8 is
6      Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
7             B : in  STD_LOGIC_VECTOR (7 downto 0);
8             S : out  STD_LOGIC_VECTOR (15 downto 0)
9           );
10 end Moltiplicatore_8;
11
12 architecture Behavioral of Moltiplicatore_8 is
13
14 component ProdottoParziale is
15    Port (  X : in  STD_LOGIC_VECTOR (7 downto 0);
16            Y : in  STD_LOGIC;
17            Z : out  STD_LOGIC_VECTOR (7 downto 0)
18          );
19 end component;
20
21 component CarrySelect_8 is
22     Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
23            B : in  STD_LOGIC_VECTOR (7 downto 0);
24            Cin : in  STD_LOGIC;
25            S : out  STD_LOGIC_VECTOR (7 downto 0);
26            Cout : out  STD_LOGIC
27          );
28 end component;
29    --segnali di suppoorto per collegare tra di loro le uscite dei parziali con i sommatori
30 signal Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7 : STD_LOGIC_VECTOR (7 downto 0);
31    --segnali di supporto dell'uscita dei sommatori per collegarli tra di loro
32 signal W0, W1, W2, W3, W4, W5, W6 : STD_LOGIC_VECTOR (7 downto 0);
33    --segnali di supporto per collegare i riporti ai rispettivi ingressi dei soomatori
34 signal Cout0, Cout1, Cout2, Cout3, Cout4, Cout5, Cout6 : STD_LOGIC;
35    --segnali di supporto per creare il segnali in ingresso al sommatore
36 signal yyy0,yyy1,yyy2,yyy3,yyy4,yyy5,yyy6 : STD_LOGIC_VECTOR (7 downto 0);
37
```

```
37
38  begin
39
40     --calcolo dei prodotti parziali
41     Parz0 : ProdottoParziale port map (X=>A, Y=>B(0), Z=>Z0);
42     Parz1 : ProdottoParziale port map (X=>A, Y=>B(1), Z=>Z1);
43     Parz2 : ProdottoParziale port map (X=>A, Y=>B(2), Z=>Z2);
44     Parz3 : ProdottoParziale port map (X=>A, Y=>B(3), Z=>Z3);
45     Parz4 : ProdottoParziale port map (X=>A, Y=>B(4), Z=>Z4);
46     Parz5 : ProdottoParziale port map (X=>A, Y=>B(5), Z=>Z5);
47     Parz6 : ProdottoParziale port map (X=>A, Y=>B(6), Z=>Z6);
48     Parz7 : ProdottoParziale port map (X=>A, Y=>B(7), Z=>Z7);
49
50     --collego i sommatori con i prodotti parziali per generare l'uscita
51
52        yyy0 <= '0' & Z0(7 downto 1);           --segnale in ingresso al primo sommatore
53     Sommatore0 : CarrySelect_8 port map (A=>yyy0, B=>Z1, Cin=>'0', S=>W0, Cout=>Cout0);
54        yyy1 <= Cout0 & W0(7 downto 1);         --segnale in ingresso al secondo sommatore
55     Sommatore1 : CarrySelect_8 port map (A=>yyy1, B=>Z2, Cin=>'0', S=>W1, Cout=>Cout1);
56        yyy2 <= Cout1 & W1(7 downto 1);         --segnale in ingresso al terzo sommatore
57     Sommatore2 : CarrySelect_8 port map (A=>yyy2, B=>Z3, Cin=>'0', S=>W2, Cout=>Cout2);
58        yyy3 <= Cout2 & W2(7 downto 1);         --segnale in ingresso al quarto sommatore
59     Sommatore3 : CarrySelect_8 port map (A=>yyy3, B=>Z4, Cin=>'0', S=>W3, Cout=>Cout3);
60        yyy4 <= Cout3 & W3(7 downto 1);         --segnale in ingresso al quinto sommatore
61     Sommatore4 : CarrySelect_8 port map (A=>yyy4, B=>Z5, Cin=>'0', S=>W4, Cout=>Cout4);
62        yyy5 <= Cout4 & W4(7 downto 1);         --segnale in ingresso al sesto sommatore
63     Sommatore5 : CarrySelect_8 port map (A=>yyy5, B=>Z6, Cin=>'0', S=>W5, Cout=>Cout5);
64        yyy6 <= Cout5 & W5(7 downto 1);         --segnale in ingresso al settimo sommatore
65     Sommatore6 : CarrySelect_8 port map (A=>yyy6, B=>Z7, Cin=>'0', S=>W6, Cout=>Cout6);
66
67     --collego le uscite dei sommatori con l'uscita generale del moltiplicatore
68
69     S(0)<=Z0(0);
70     S(1)<=W0(0);
71     S(2)<=W1(0);
72     S(3)<=W2(0);
73     S(4)<=W3(0);
74     S(5)<=W4(0);
75     S(6)<=W5(0);
76     S(14 downto 7)<=W6;
77     S(15)<=Cout6;
78
79  end Behavioral;
80
```

Figura 3.3 – *Implementazione in VHDL del moltiplicatore Ripple-Carry ad Array.*

Dal listato mostrato in Figura 3.3 si può vedere come sia stato realizzato il moltiplicatore in VHDL. Inizialmente, utilizzando il componente 'ProdottoParziale' si realizzano di seguito i prodotti parziali (AND tra gli ingressi X_i e Y_j) successivamente, utilizzando i segnali di supporto, si collegano opportunamente i vari sommatori tra di loro, come mostrato nello schema di principio (sono stati utilizzati dei Carry-Select a 8 bit).
Questo sommatore è scarsamente ottimizzato, infatti il carry di ingresso di ogni sommatore è posto a livello logico 0. Inoltre, si può dimostrare che, il tempo di ritardo del moltiplicatore è di ordine o(3xN). Bisogna quindi creare una struttura in grado di ridurre il tempo di propagazione di 3xN. È sufficiente modificare la struttura interna facendo in modo che il riporto non venga propagato lungo la riga, bensì venga "inviato" alla cella più in basso.

3.2 Moltiplicatore Carry-Save

La seconda architettura hardware, ovvero moltiplicatore parallelo, viene mostrata in Figura 3.4. Partiamo sempre da una struttura più semplice a 4 bit per arrivare successivamente ad un moltiplicatore ad 8 bit.

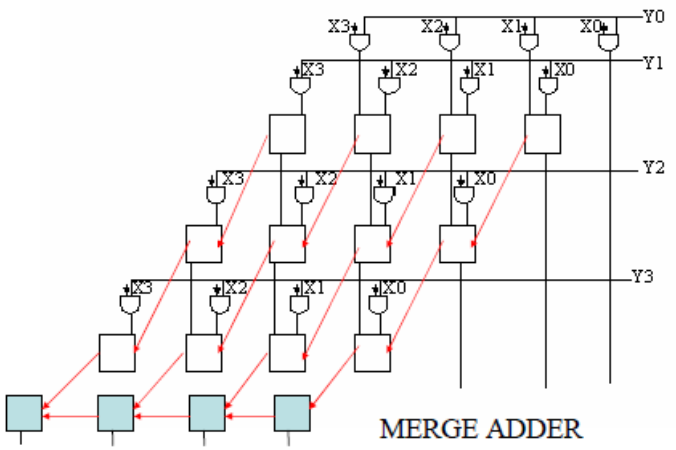

Figura 3.4 – *Schema elettrico di principio di un moltiplicatore Carry-Save a 4 bit (http://www.unibo.it/, "Elettronica dei Sistemi Digitali LA").*

Dallo schema di principio si può capire che il bit di riporto non "viaggia" più sulla riga ma viene inviato alla cella più in basso a sinistra sulla stessa riga (termini che hanno lo stesso peso). Si noti che è stato aggiunto un Merge-Adder (ovvero un qualsiasi sommatore studiato in precedenza) che somma gli N bit più significativi, mentre gli altri N bit vengono ottenuti immediatamente dalle celle precedenti.
Successivamente, utilizzando lo schema di Figura 3.4, è possibile realizzare un moltiplicatore Carry-Save ad 8 bit.

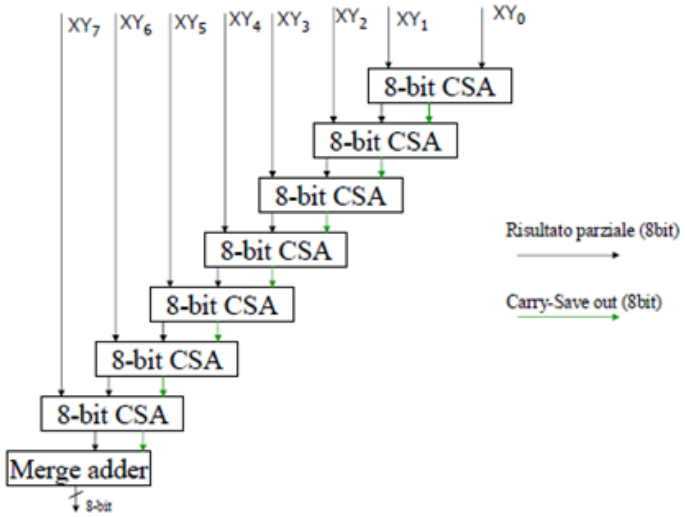

Figura 3.5 – *Schema di principio di un moltiplicatore Carry-Save a 8 bit (http://www.unibo.it/, "Elettronica dei Sistemi Digitali LA").*

A questo punto, utilizzando lo schema di Figura 3.5, è possibile implementare in VHDL un moltiplicatore Carry-Save ad 8 bit.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity Moltiplicatore_CarrySave_8 is
6      Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
7             B : in  STD_LOGIC_VECTOR (7 downto 0);
8             S : out  STD_LOGIC_VECTOR (15 downto 0));
9  end Moltiplicatore_CarrySave_8;
10
11 architecture Behavioral of Moltiplicatore_CarrySave_8 is
12
13 component ProdottoParziale is
14    Port (  X : in  STD_LOGIC_VECTOR (7 downto 0);
15            Y : in  STD_LOGIC;
16            Z : out  STD_LOGIC_VECTOR (7 downto 0)
17          );
18 end component;
19
20 component CarrySelect_8 is
21     Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
22            B : in  STD_LOGIC_VECTOR (7 downto 0);
23            Cin : in  STD_LOGIC;
24            S : out  STD_LOGIC_VECTOR (7 downto 0);
25            Cout : out  STD_LOGIC
26           );
27 end component;
28
29 component CarrySave_8
30     Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
31            B : in  STD_LOGIC_VECTOR (7 downto 0);
32            Cin : in  STD_LOGIC_VECTOR (7 downto 0);
33            S : out  STD_LOGIC_VECTOR (7 downto 0);
34            Cout : out  STD_LOGIC_VECTOR (7 downto 0)
35           );
36 end component;
37
38    --segnali di suppoorto per collegare tra di loro le uscite dei parziali con i sommatori
39 signal Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7 : STD_LOGIC_VECTOR (7 downto 0);
40    --segnali di supporto dell'uscita dei sommatori per collegarli tra di loro
41 signal W0, W1, W2, W3, W4, W5, W6, W7 : STD_LOGIC_VECTOR (7 downto 0);
42    --segnali di supporto per collegare i riporti ai rispettivi ingressi dei soomatori
43 signal Cout0, Cout1, Cout2, Cout3, Cout4, Cout5, Cout6: STD_LOGIC_VECTOR (7 downto 0);
44    --segnali di supporto per creare il segnali in ingresso al sommatore
45 signal yyy0,yyy1,yyy2,yyy3,yyy4,yyy5,yyy6,yyy7 : STD_LOGIC_VECTOR (7 downto 0);
46 signal res : STD_LOGIC; --segnale di supporto per il resto del mergeAdder che sarà sempre zero
47
48 begin
49
50    --creo la matrice dei prodotti parziali
51
52    Parz0 : ProdottoParziale port map (X=>A, Y=>B(0), Z=>Z0);
53    Parz1 : ProdottoParziale port map (X=>A, Y=>B(1), Z=>Z1);
54    Parz2 : ProdottoParziale port map (X=>A, Y=>B(2), Z=>Z2);
55    Parz3 : ProdottoParziale port map (X=>A, Y=>B(3), Z=>Z3);
56    Parz4 : ProdottoParziale port map (X=>A, Y=>B(4), Z=>Z4);
57    Parz5 : ProdottoParziale port map (X=>A, Y=>B(5), Z=>Z5);
58    Parz6 : ProdottoParziale port map (X=>A, Y=>B(6), Z=>Z6);
59    Parz7 : ProdottoParziale port map (X=>A, Y=>B(7), Z=>Z7);
60
61    --collego i sommatori con i prodotti parziali per generare l'uscita
62
63       yyy0 <= '0' & Z0(7 downto 1);          --segnale in ingresso al primo sommatore
64    CS_0 : CarrySave_8 port map (A=>yyy0, B=>Z1, Cin="00000000", S=>W0, Cout=>Cout0);
65       yyy1 <= '0' & W0(7 downto 1);          --segnale in ingresso al secondo sommatore
66    CS_1 : CarrySave_8 port map (A=>yyy1, B=>Z2, Cin=>Cout0, S=>W1, Cout=>Cout1);
67       yyy2 <= '0' & W1(7 downto 1);          --segnale in ingresso al terzo sommatore
68    CS_2 : CarrySave_8 port map (A=>yyy2, B=>Z3, Cin=>Cout1, S=>W2, Cout=>Cout2);
69       yyy3 <= '0' & W2(7 downto 1);          --segnale in ingresso al quarto sommatore
70    CS_3 : CarrySave_8 port map (A=>yyy3, B=>Z4, Cin=>Cout2, S=>W3, Cout=>Cout3);
71       yyy4 <= '0' & W3(7 downto 1);          --segnale in ingresso al quinto sommatore
72    CS_4 : CarrySave_8 port map (A=>yyy4, B=>Z5, Cin=>Cout3, S=>W4, Cout=>Cout4);
73       yyy5 <= '0' & W4(7 downto 1);          --segnale in ingresso al sesto sommatore
74    CS_5 : CarrySave_8 port map (A=>yyy5, B=>Z6, Cin=>Cout4, S=>W5, Cout=>Cout5);
75       yyy6 <= '0' & W5(7 downto 1);          --segnale in ingresso al settimo sommatore
76    CS_6 : CarrySave_8 port map (A=>yyy6, B=>Z7, Cin=>Cout5, S=>W6, Cout=>Cout6);
77       yyy7 <= '0' & W6(7 downto 1);          --segnale in ingresso al merge adder
78    MergeAdder : CarrySelect_8 port map (A=>yyy7, B=>Cout6, Cin=>'0', S=>W7, Cout=>res);
79             --Sommo  l'ultima risultato con l'ultimo resto usando un sommatore Carryselect
80
81    --collego le uscite dei sommatori con l'uscita generale del moltiplicatore
82
83    S(0)<=Z0(0);
84    S(1)<=W0(0);
85    S(2)<=W1(0);
86    S(3)<=W2(0);
87    S(4)<=W3(0);
88    S(5)<=W4(0);
89    S(6)<=W5(0);
90    S(7)<=W6(0);
91    S(15 downto 8)<=W7;
92
93 end Behavioral;
94
```

Figura 3.6 – *Implementazione in VHDL del moltiplicatore Carry-Save ad 8 bit.*

Moltiplicatori a 8 bit

La prima operazione da eseguire per realizzare il moltiplicatore in VHDL sta nel creare la matrice dei prodotti parziali. Utilizzando quindi, anche in questo caso, il componente 'ProdottoParziale' si ottengono i prodotti parziali che successivamente verranno sommati a coppie utilizzando un sommatore Carry-Save-Adders. Questo sommatore non è altro che un insieme di otto Full-Adders raggruppati in un sottoblocco che in uscita riporta due bus da 8 bit: uno per il segnale somma (S) e uno per il segnale riporto (C_{out}).
Si può dimostrare chi il ritardo di calcolo per questo moltiplicatore è dell'ordine di o(N).
Per migliorare ulteriormente le prestazioni dei moltiplicatori e ridurre il tempo di calcolo è necessario modificare nuovamente la struttura per arrivare quindi ad una struttura logaritmica. Questo nuovo moltiplicatore prenderà il nome di "moltiplicatore di Wallace".

Il moltiplicatore di Wallace è una particolare ottimizzazione del moltiplicatore Carry-Save studiata per ottenere la minima propagazione del ritardo. La struttura dell'albero è mostrata in Figura 4.1:

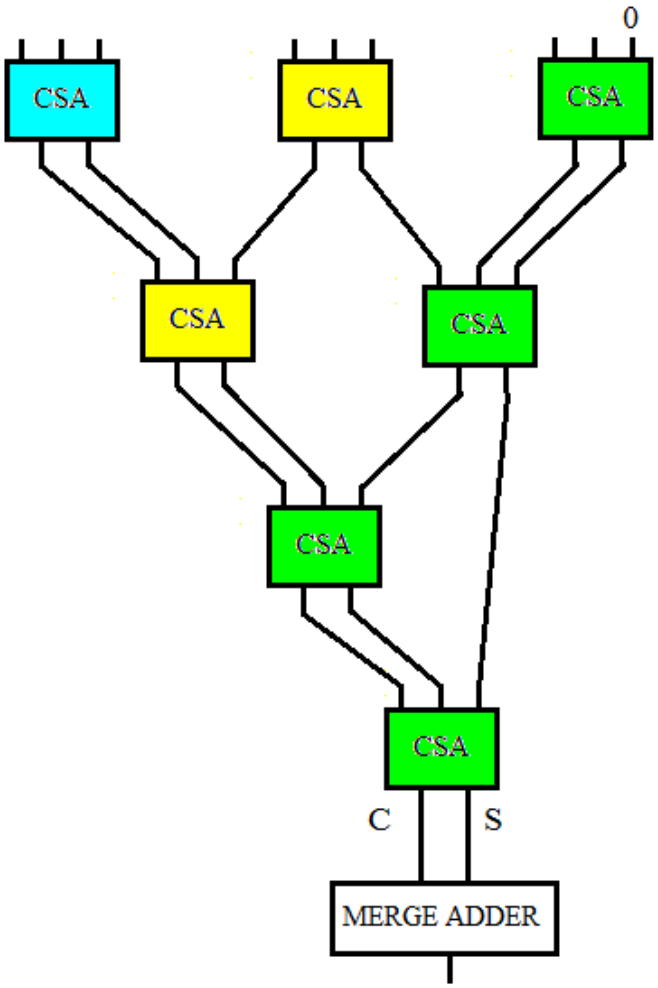

Figura 4.1 – *Schema di principio di un moltiplicatore di Wallace a 8 bit*
(Tesi di Laurea, "Circuiti Digitali a Supporto di Moltiplicazioni fra Numeri Interi Grandi").

Invece di sommare i prodotti parziali a coppie, il moltiplicatore di Wallace somma tutti i bit con lo stesso peso in un unico albero. L'albero di Wallace rappresento dalla Figura 4.1 è composto da un insieme di Carry-Save-Adders (CSA). A sua volta ogni CSA è costituito da Full-Adders, dove il bit di carry viene "portato fuori" anziché essere collegato al prossimo bit più significativo. Questo comporta che 3 bit dello stesso peso vengono sommati per ottenere 2 bit, ovvero: il bit di carry con peso N+1 e il bit di somma con peso N. A questo punto si evince, quindi, che ogni strato dell'albero riduce la dimensione dei vettori di un fattore 3:2. L'albero avrà tanti strati quanti sono quelli necessari per ridurre il numero di vettori a due (somma più carry). Un sommatore (Merge-Adder della Figura 4.1) studiato in precedenza combinerà questi vettori per ottenere il risultato finale.

4.1 Implementazione e simulazione in VHDL

L'implementazione in VHDL del moltiplicatore di Wallace viene mostrata nella Figura 4.2:

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4
5   entity Moltiplicatore_Wallace_8 is
6       Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
7              B : in  STD_LOGIC_VECTOR (7 downto 0);
8              S : out  STD_LOGIC_VECTOR (15 downto 0));
9   end Moltiplicatore_Wallace_8;
10
11  architecture Behavioral of Moltiplicatore_Wallace_8 is
12
13  component ProdottoParziale is
14     Port (  X : in  STD_LOGIC_VECTOR (7 downto 0);
15             Y : in  STD_LOGIC;
16             Z : out  STD_LOGIC_VECTOR (7 downto 0)
17           );
18  end component;
19
20  component CarrySelect_16 is
21      Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
22             B : in  STD_LOGIC_VECTOR (15 downto 0);
23             Cin : in  STD_LOGIC;
24             S : out  STD_LOGIC_VECTOR (15 downto 0);
25             Cout : out  STD_LOGIC
26            );
27  end component;
28
29  component CarrySave_16
30      Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
31             B : in  STD_LOGIC_VECTOR (15 downto 0);
32             Cin : in  STD_LOGIC_VECTOR (15 downto 0);
33             S : out  STD_LOGIC_VECTOR (15 downto 0);
34             Cout : out  STD_LOGIC_VECTOR (15 downto 0)
35            );
36  end component;
37
38     --segnali di suppoorto per collegare tra di loro le uscite dei parziali con i sommatori
39  signal Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7 : STD_LOGIC_VECTOR (7 downto 0);
40     --segnali di supporto per creare la matrice della moltiplicazione con i giusti pesi
41  signal Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7 : STD_LOGIC_VECTOR (15 downto 0);
42     --segnali di supporto dell'uscita dei sommatori per collegarli tra di loro
43  signal W00, W01, W02, W10, W11, W20, W30 : STD_LOGIC_VECTOR (15 downto 0);
44     --segnali di supporto per collegare i riporti ai rispettivi ingressi dei soomatori
45  signal Cout00, Cout01, Cout02, Cout10, Cout11, Cout20, cout30 : STD_LOGIC_VECTOR (15 downto 0);
46     --segnali di supp per segnali in ingresso al sommatore successivo (per dare il giusto peso al carry)
47  signal yyy0, yyy1, yyy2, yyy3, yyy4, yyy5, yyy6 : STD_LOGIC_VECTOR (15 downto 0);
48  signal res : STD_LOGIC; --segnale di supporto per il resto del mergeAdder che sarà sempre zero
49
50  begin
51
52  --calcolo i prodotti parziali
53
54     Parz0 : ProdottoParziale port map (X=>A, Y=>B(0), Z=>Z0);
55     Parz1 : ProdottoParziale port map (X=>A, Y=>B(1), Z=>Z1);
56     Parz2 : ProdottoParziale port map (X=>A, Y=>B(2), Z=>Z2);
57     Parz3 : ProdottoParziale port map (X=>A, Y=>B(3), Z=>Z3);
58     Parz4 : ProdottoParziale port map (X=>A, Y=>B(4), Z=>Z4);
59     Parz5 : ProdottoParziale port map (X=>A, Y=>B(5), Z=>Z5);
60     Parz6 : ProdottoParziale port map (X=>A, Y=>B(6), Z=>Z6);
61     Parz7 : ProdottoParziale port map (X=>A, Y=>B(7), Z=>Z7);
62
63  --creo la matrice dei prodotti parziali con i giusti pesi
64
65     Y0 <= "00000000" & Z0;
66     Y1 <= "0000000" & Z1 & "0";
67     Y2 <= "000000" & Z2 & "00";
68     Y3 <= "00000" & Z3 & "000";
69     Y4 <= "0000" & Z4 & "0000";
70     Y5 <= "000" & Z5 & "00000";
71     Y6 <= "00" & Z6 & "000000";
72     Y7 <= "0" & Z7 & "0000000";
73
74  --primo strato dell'albero
75
76     Sommatore0_0 : CarrySave_16 port map (A=>Y0, B=>Y1, Cin=> (others =>'0'), S=>W00, Cout=>Cout00);
77     Sommatore0_1 : CarrySave_16 port map (A=>Y2, B=>Y3, Cin=>Y4, S=>W01, Cout=>Cout01);
78     Sommatore0_2 : CarrySave_16 port map (A=>Y5, B=>Y6, Cin=>Y7, S=>W02, Cout=>Cout02);
79
80  --secondo strato dell'albero
81
82     Sommatore1_0 : CarrySave_16 port map (A=>W00, B=>W01, Cin=>W02, S=>W10, Cout=>Cout10);
83        yyy0 <= Cout00 (14 downto 0) & '0'; --do il giusto peso al carry
84        yyy1 <= Cout01 (14 downto 0) & '0'; --do il giusto peso al carry
85        yyy2 <= Cout02 (14 downto 0) & '0'; --do il giusto peso al carry
86     Sommatore1_1 : CarrySave_16 port map (A=>yyy0, B=>yyy1, Cin=>yyy2, S=>W11, Cout=>Cout11);
87
88  --terzo strato dell'albero
89
90        yyy3 <= Cout10 (14 downto 0) & '0';
91        yyy4 <= cout11 (14 downto 0) & '0';
92     Sommatore2_0 : CarrySave_16 port map (A=>W11, B=>yyy3, Cin=>yyy4, S=>W20, Cout=>Cout20);
93
94  --quarto strato dell'albero
95
96        yyy5 <= Cout20 (14 downto 0) & '0';
97     Sommatore3_0 : CarrySave_16 port map (A=>W20, B=>W10, Cin=>yyy5, S=>W30, Cout=>Cout30);
98
99  --calcolo la somma finale
100
101       yyy6 <= Cout30 (14 downto 0) & '0';
102       MergeAdder : CarrySelect_16 port map (A=>W30, B=>yyy6, Cin=>'0', S=>S, Cout=>res);
103
104 end Behavioral;
```

Figura 4.2 – *Implementazione in VHDL del moltiplicatore di Wallace a 8 bit.*

La simulazione del moltiplicatore di Wallace viene mostrata nella Figura 4.3:

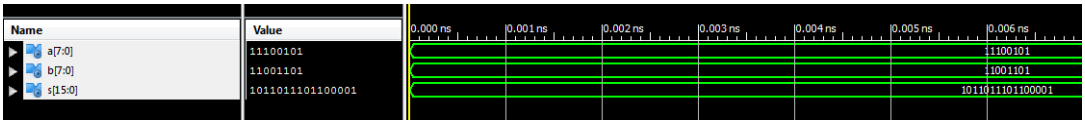

Figura 4.3 – *Simulazione di una moltiplicazione utilizzando il moltiplicatore di Wallace.*

Per implementare il moltiplicatore di Wallace in VHDL per prima cosa si inizializzano i componenti necessari, ovvero 'ProdottoParziale', 'Carry-Select da 16 bit', 'Carry-Save da 16 bit'.
Per eseguire il corpo del programma è stato seguito lo schema mostrato in Figura 4.1. Per prima cosa sono stati calcolati i prodotti parziali, poi è stata creata la matrice degli stessi prodotti parziali giustamente pesata e infine sono stati collegati i vari strati dell'albero utilizzando dei sommatori Carry-Save-Adders. La moltiplicazione finale viene data sommando l'ultima somma (S) con l'ultimo carry (C_{out}) utilizzando un sommatore Carry-Select da 16 bit.

4.2 Considerazioni sul moltiplicatore di Wallace

Il moltiplicatore di Wallace è costituito dallo stesso numero di sommatori Carry-Save-Adders che si utilizzano per il moltiplicatore Carry-Save. Il moltiplicatore comunque riordina le connessioni in modo tale che i bit dei prodotti parziali, che hanno ritardi maggiori, siano connessi più vicino alla radice dell'albero.
Questa nuova riorganizzazione cambia il ritardo da ordine o(N) ad ordine o(log(N)).
Si noti che nella simulazione sono stati utilizzati dei sommatori Carry-Save-Adders da 16 bit (quindi 16 Full-Adders), questo per non incorrere in problemi di overflow. Di conseguenza, se si passasse alla realizzazione hardware avremmo un maggior costo in termini di risorse ma un notevole vantaggio in termini di tempo di calcolo.

Conclusioni

In questo elaborato è stato realizzato il moltiplicatore di Wallace simulandolo nel linguaggio VHDL.
Particolare attenzione è stata posta nell'implementare le varie parti del moltiplicatore di Wallace in VHDL ottenendo quindi la simulazione finale della moltiplicazione tra due parole di 8 bit ciascuna.
La simulazione del moltiplicatore di Wallace dimostra la corretta funzionalità del circuito, a verifica quindi della giusta applicazione di tutti i componenti che formano il moltiplicatore.
Dal lavoro svolto nel realizzare i vari "blocchi" che costituiscono il moltiplicatore, si è potuto inoltre apprendere una caratteristica distintiva del linguaggio VHDL: ovvero la possibilità di modellare facilmente l'interazione dei vari blocchi funzionali che compongono un sistema.
Per rendere completa la simulazione bisognerebbe verificare anche i tempi di calcolo dei singoli componenti studiati, parte che non è stata approfondita in quanto l'obiettivo primario del trattato risulta essere la focalizzazione all'apprendimento del linguaggio VHDL.

Elenco delle figure

Ringraziamenti

A conclusione dei miei anni di università, colgo l'occasione per ringraziare vivamente le persone che da sempre mi sono state vicine sia nell'incoraggiamento allo studio che nel sostegno morale.
Innanzitutto vorrei ringraziare il Professor Simone Buso, egregio stimolatore e sempre disponibile a fornire consigli e utili chiarimenti.
Un sentito ringraziamento a tutti i miei compagni di avventura e agli amici che mi hanno dimostrato una sincera testimonianza di vita.
In particolare ringrazio i miei genitori unitamente alla sorella Laura e al cognato Michele per quanto fatto in questo periodo dedicato alla facoltà di Ingegneria.

Bibliografia

S.Buso, appunti del corso “Teoria dei Circuiti Digitali”, 2009-2010

John F.Wakerly, “Digital Design Principle and Practice (Fourth Edition)”, Prentice Hall, 2005

C.Brandolese, http://home.deib.polimi.it/, “Introduzione al linguaggio VHDL, Aspetti teorici ed esempi di progettazione” ultimo accesso 20/09/14

C.Giaconia, http://www.unipa.it/, “Corso di Architettura dei Sistemi Integrati-Note sul VHDL” ultimo accesso 20/09/14

S.Ferrari, http://homes.di.unimi.it/, “Elementi di VHDL”, ultimo accesso 20/09/14

G.Ramponi, http://www2.units.it/, “I Circuiti sommatori”, ultimo accesso 20/09/14

F.Campi, http://www.unibo.it/, “Elettronica dei Sistemi Digitali LA”, versione 2005-2006

D.Susino, Tesi di Laurea, “Circuiti Digitali a Supporto di Moltiplicazioni fra Numeri Interi Grandi” Anno Accademico 2007-2008