Università della Calabria

Progetto Elettronica Digitale

Francesca Vaccaro 239641
Luca Timpano 240236
Greta Tigano 239619

Instructor: Stefania Perri
Corso di Studio in Ingegneria Informatica

2024/2025

Contents

1 Introduzione

Il seguente progetto è finalizzato all'analisi della progettazione di un Carry Select Adder (CSA). A differenza dei tradizionali addizionatori il CSA si propone per ridurre i tempi di attesa associata alla propagazione del carry(riporto), ottenendo migliori prestazioni complessive. Nel nostro caso andremo a progettare un CSA a 16 bit, esaminando la sua architettura e il funzionamento dei componenti chiave.
L'analisi teorica che seguirà si occuperà di fornire una base solida per la comprensione dei circuiti sommatori, partendo dal Full Adder fino all'implementazione del Carry Select Adder.

2 Analisi Teorica

2.1 Full Adder (FA)

Il full adder è un circuito combinatorio deputato all'operazione di somma tra tre bit: due bit di ingresso A e B e il *carry-in*, ossia il bit di riporto generato dalla somma precedente. In output restituisce il bit di somma S, e il riporto da utilizzare nella successiva operazione: *carry-out*. Il full adder è una versione estesa dell'*half adder*, che processa la somma solo tra due bit. Per la realizzazione del full adder è fondamentale tenere conto della sua **tabella di verità**:

Inputs			Outputs	
A	B	C_{in}	Sum	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

Table 1: Tabella degli input e output di un full adder

2.1.1 Le mappe di Karnaugh

Volendo riscrivere la tabella di verità sotto forma di mappe di Karnaugh otteniamo:

A \ BCin	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Figure 1: Mappa di Karnaugh per C_{out}

A \ BCin	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Figure 2: Mappa di Karnaugh per Sum

Ottenendo le formule:

$$\boxed{C_{\text{out}} = B \cdot C_{\text{in}} + A \cdot C_{\text{in}} + A \cdot B} \tag{1}$$

$$S = \overline{A} \cdot \overline{B} \cdot \overline{Ci} + \overline{A} \cdot B \cdot \overline{Ci} + A \cdot \overline{B} \cdot \overline{Ci} + A \cdot B \cdot \overline{Ci}$$

$$= \overline{A} \cdot (\overline{B} \cdot Ci + B \cdot Ci) + A \cdot (\overline{B} \cdot Ci + B \cdot Ci)$$

$$= \overline{A} \cdot (B \oplus Ci) + A \cdot (B \oplus Ci)$$

Ponendo $Y = B \oplus Ci$

$$\boxed{S = \overline{A} \cdot Y + A \cdot \overline{Y} = A \oplus B \oplus Ci}$$

Queste formule, sebbene corrette, risultano piuttosto costose e inutilmente complesse da implementare, nel paragrafo successivo andremo a ricavare, attraverso l'analisi della tabella di verità, le espressioni booleane con cui andremo a implementare il full adder.

2.1.2 Le espressioni booleane

L'implementazione del full adder avviene attraverso l'uso di *porte logiche* come l'**AND**, l'**OR** e lo **XOR**.

- **Somma(S)**: La somma viene generata utilizzando la porta XOR, poichè corrisponde alla somma binaria tra due bit.
- **Carry-Out (C_out)** il bit di riporto viene invece generato quando la somma produce un valore che richiede un riporto al bit successivo. Consideriamo i tre ingressi A, B e C_{in}, il carry-out si verificherà in due situazioni principali:
  - Se il valore di A e il valore B sono entrambi corrispondenti a 1, la loro somma produce un riporto indipendentemente dal valore di C_{in}
  - Se uno tra A e B è 1, e anche C_{in} è uguale a 1, allora si genererà un carry-out

  La generazione del carry-out dipenderà quindi dal verificarsi di almeno due delle tre seguenti condizioni:
  - A = 1

- B = 1
- C = 1

Alla luce di queste premesse possiamo quindi ricavarci le equazioni booleane del bit somma e del carry-out:

$$\text{Sum} = A \oplus B \oplus C_{\text{in}} \tag{2}$$

$$C_{\text{out}} = (A \cdot B) + (A \cdot C_{\text{in}}) + (B \cdot C_{\text{out}}) \tag{3}$$

Questo schema consente di concatenare più Full Adders per sommare numeri a più bit; il bit carry-out generato da un Full Adder diventa il carry-in del Full Adder successivo, permettendo così di gestire l'overflow durante la somma. Si anticipa che il segnale C_{in} può essere anche espresso mediante l'uso di due segnali:

- Generate
- Propagate

$$C_{\text{out}} = A \cdot B + (A \oplus B) \cdot C_{\text{in}} \tag{4}$$

dove:

- $A \cdot B$ rappresenta il *generate*
- $A \oplus B$ rappresenta il *propagate*

Nel paragrafo 3.1 tratteremo la progettazione di un full adder mediante l'uso di questi due segnali.

2.1.3 L'importanza del Full Adder

Il Full Adder è un componente fondamentale nei circuti di somma. Si implementa in architetture leggermente più avanzate come il **ripple carry** e il **carry select**. Successivamente verranno illustrate le tecniche per rendere più efficiente l'operazione di somma

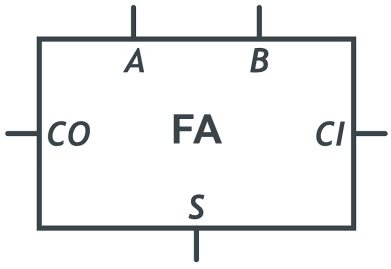


Figure 3: Schema di un Full Adder

2.2 Ripple Carry Adder (RCA)

2.2.1 Importanza della somma

All'interno di un calcolatore, la somma ha una notevole importanza, in quanto tutte le altre operazioni che vengono richieste si possono esprimere in funzione di quest'ultima.
Come trattato nel precedente paragrafo, per eseguire la somma ci avvaliamo dell'elemento Full Adder, ma questo non basta, in quanto non può singolarmente occuparsi di somme composte da più bit.
Abbiamo dunque bisogno di mettere insieme più full adder, in modo da fare la somma a più bit, avremo mediamente bisogno di tanti full adder quanti sono i bit interessati, ovviamente però ci sono casi particolari e aggiunte, che richiederanno l'utilizzo di strutture più articolate.

2.2.2 Introduzione al RCA

Una prima architettura che ci viene in aiuto è proprio il **Ripple Carry Adder**, che esplica già nel nome il suo funzionamento, si parlerà infatti di "*propagazione di riporto*", in quanto il carry generato da ciascun full adder verrà dato in ingresso al full adder successivo, questa operazione si ripete fino all'ultimo full adder; allo stesso tempo la "S" che ogni full adder produce, quindi la somma senza riporto, verrà mandata in output.
Un aspetto significativo si osserva nell'ultimo full adder della catena, cioè quello relativo al bit più significativo: qui, infatti, il riporto in uscita (C_n) rappresenta il bit più significativo del risultato finale.
Questa affermazione appena data seppur vera non è del tutto completa, difatti questo vale solo nel caso in cui stiamo andando a trattare dei numeri cosidetti "*unsigned*", ovvero senza segno; se li trattassimo con segno allora dovremmo prendere in considerazione quale metodologia di rappresentazione utilizzare, se "*complemento a 2*" oppure quella "*modulo e segno*".
Per fare un semplice esempio, prendiamo la seguente sequenza di bit:

1 1 1 1

Se lo trattassimo come **unsigned** avremmo il numero in decimale "15".
Quando invece è rappresentato in **modulo e segno**, bisogna prendere il bit più significativo che indicherà il segno, se è 0 allora il numero è positivo, se è 1 è negativo, e poi prendo la restante parte della sequenza e la porto in notazione decimale, in questo caso vale 7; il risultato finale sarà "-7".
L'ultima rappresentazione che andiamo ad analizzare è quella in **complemento a 2**, che a livello di procedura potrebbe risultare leggermente più articolata; anche qui vado a prendere il segno che corrisponde al bit più significativo, questa volta però includo anche il bit del segno per la notazione in decimale, ottenendo nel nostro caso -8, il passo successivo è quello di escludere il bit più significativo e trasformare solo l'ultima parte in decimale, ottenendo quindi 7, l'ultimo passaggio consiste nel fare la somma tra i due numeri ottenuti, quindi il risultato finale sarà "-1".
Si noti dunque l'importanza di conoscere la rappresentazione che si sta andando ad utilizzare, il risultato potrebbe essere totalmente diverso.

2.2.3 Rappresentazione grafica della struttura

Vediamo come è logicamente organizzata l'architettura, attraverso l'utilizzo dei full adder, e rappresentiamo un rca a 4 bit, dunque mediante l'implementazione dei 4 full adder:

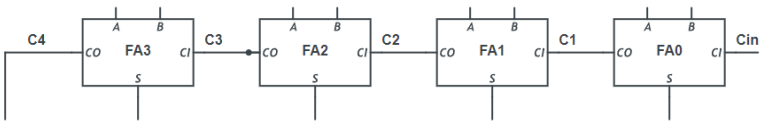


Figure 4: Schema di un Ripple Carry a 4 bit

Attraverso la versione grafica di questo ripple carry a 4 bit, possiamo notare ancora meglio il suo funzionamento(anche se ancora superficialmente), e come i bit di riporto vengano passati in ingresso al full adder successivo.

2.2.4 Punti deboli del ripple carry

Proprio per via della propagazione del riporto, emergono le maggiori debolezze del ripple carry, difatti se si dovesse analizzare in maniera approfondita l'architettura, ci renderemmo conto che per far lavorare FA1 serve per forza il bit di riporto C_1, per FA2 ci serve C_2 e così via; si noti dunque che andiamo a perdere molto in termini di velocità
Supponiamo che il full adder impieghi un tempo τ_{carry} che corrisponde a 100 ps, utilizzando il ripple carry adder appena costruito, quindi a 4 bit, al tempo totale impiegato sarà $4 \cdot \tau_{carry}$, si noti che in un microprocessore odierno a 64 bit otterremmo una frequenza di circa 6400 ps che è 20 volte minore di quelle che si utilizzano oggi(4 Ghz); nelle nostre architetture dunque, non viene utilizzato il ripple carry adder.

2.3 Carry Select Adder (CSA)

Come precedentemente anticipato, quando lavoriamo nel campo dei circuiti digitali, l'addizione di numeri binari è una delle operazioni più importanti, ed è per questo che il **Carry Select Adder (CSA)** è una soluzione progettata per migliorare l'operazione di somma cercando di ridurre il tempo di attesa per la propagazione del *carry*.
Solitamente, sommando due numeri binari, si genera un riporto che viene propagato al bit successivo, logica che è alla base della *Ripple Carry Adder* (RCA), per cui ogni *full adder* deve necessariamente attendere il bit di riporto precedente per completare la somma. Questo, ovviamente, può implicare un fattore di ritardo nei circuiti con tante cifre binarie, dunque, spesso si preferisce utilizzare proprio il *Carry Select Adder*.

Tale addizionatore aritmetico fa in modo che il calcolo della somma possa essere suddiviso in blocchi e ciascun blocco determina somme parallele, una assumendo che il bit del *carry-in* sia 0 e l'altra 1. Così facendo, viene ridotto notevolmente il tempo necessario per terminare la somma, perché il circuito, una volta ottenuto il riporto, va a selezionare il risultato corretto attraverso un multiplexer.
Vediamo la struttura del CSA:

2.3.1 Rappresentazione grafica e funzionamento

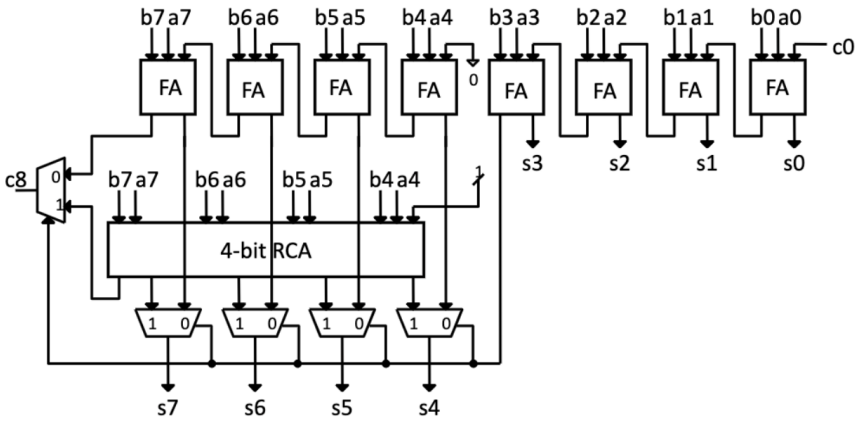


Figure 5: Schema di un Carry Select Adder

Per semplicità abbiamo rappresentato un Carry Selecet Adder a 8 bit e osserviamo che:

- **Primo blocco:** calcola la somma e il riporto in modo tradizionale.
- **Secondo blocco:** hanno due sottocomponenti che calcolano le somme parallelamente, una considerando un carry-in pari a 0, l'altra considerando che sia 1.

Quindi, nel nostro caso, ci occuperemo di un **CSA** a 16 bit, perciò avremo come primo blocco un *Ripple Carry* a 8 bit e altri due *Ripple Carry* a 8 bit che calcolano la somma parallelamente, considerando il bit 0 e il bit 1. Inoltre sfruttiamo 8 multiplexer associati a ogni coppia di *Full Adder* dei due Ripple Carry. Infine, viene utilizzato un ultimo multiplexer per gestire la scelta tra i risultati dei due Ripple Carry, considerando anche il riporto ottenuto dal primo blocco.
In questo contesto, un importante ruolo è svolto dal multiplexer, vediamo come funziona.

2.3.2 Breve descrizione del mux

Il **Multiplexer** è un componente circuitale che si occupa di selezionare uno tra i 2^n ingressi e trasmettere l'output desiderato. Proprio per questa ragione il multiplexer è anche definito *selettore*.
Osserviamo la sua struttura:

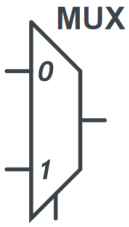


Figure 6: Schema di un MUX 2:1

Considerando il nostro caso specifico, il *Multiplexer* riceve come ingressi i risultati delle due somme precedentemente calcolate (una considerando il bit 0 e l'altra il bit 1, oltre agli ingressi riceve un segnale corrispondente al valore del carry-in del primo blocco. Una volta noto il valore effettivo del carry-in seleziona la somma corretta, che sarà l'output del multiplexer. Quindi, in conclusione, il *MUX* serve per semplificare i calcoli all'interno del Carry Select Adder.

3 Progettazione

3.1 Full Adder

Come enunciato nel paragrafo 2.1, il full adder è un circuito combinatorio fondamentale nella progettazione di sistemi digitali. Viene utilizzato per eseguire la somma di tre bit in ingresso: A,B e C_{in}. Nel seguente paragrafo tratteremo di come viene implementato in linguaggio `VHDL`. Un passaggio utile per la realizzazion e del codice è quello di visualizzare il circuito che andremo a realizzare:

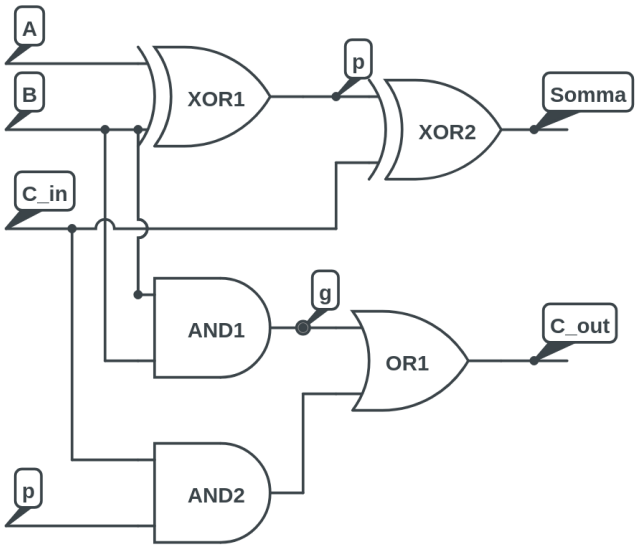


Figure 7: Porte logiche di un FA

Il codice VHDL sarà diviso in due parti principali:

- La **entity**, dove specificheremo i **port**, ovvero i segnali di input e output
- L'**architecture** dove è contenuta la descrizione del funzionamento interno del componente

Di sotto è riportato quindi il full adder descritto dalle formule 2 e 4:

```
--creiamo il fulladder
entity FullAdder is

Port(a, b, cin : in bit;          --input: a,b, carry in
    cout, sum : out bit);         --output, carry out, somma

end FullAdder;

architecture MyFa of FullAdder is

signal p,g: bit;

begin

p <= a xor b;
cout <= g or (p and cin);
sum <= p xor cin;
g <= a and b;

end MyFa;
```

Lo schematic corrispondente è il seguente:

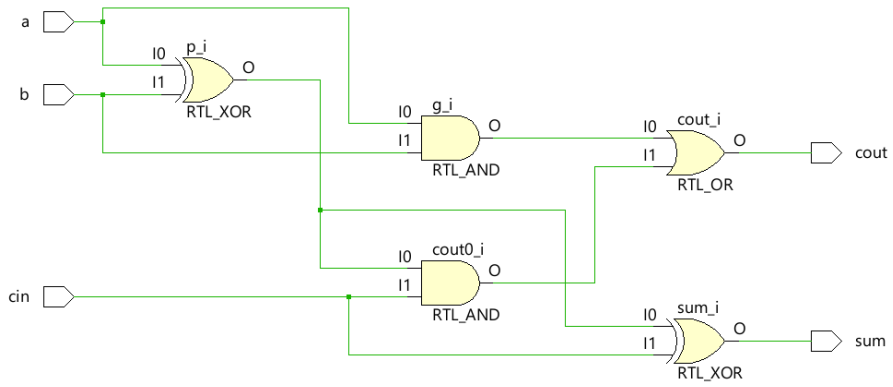


Figure 8: Schematic FA con segnale p e g

3.1.1 Full Adder con il mux

Il full adder con multiplexer è una variante del classico full adder. Il carry out viene calcolato in base al bit di selezione *propagate(p)*.

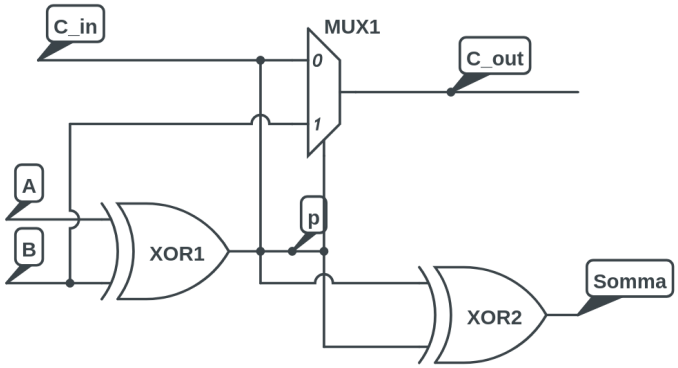


Figure 9: Porte logiche di un FA

Grazie al *MUX* possiamo semplificare il circuito sfruttando il comportamento condizionale dello stesso.

Il primo passo da eseguire è calcolare il segnale *propagate*, che indica se la somma dei bit A e B dipende dal carry-in:

$$P = A \oplus B \tag{5}$$

Abbiamo due casi:

- Se $P = 0$, allora A e B sono uguali, il carry-out viene generato indipendentemente dal carry-in.
- Se $P = 1$, allora A e B sono diversi, il carry-out dipenderà dal valore del carry-in.

```
--creiamo il fulladder
entity FullAdder is

Port(a, b, cin : in bit;
    cout, sum : out bit);

end FullAdder;

architecture MyFa of FullAdder is

signal p: bit;

begin

p <= a xor b;
--se a e b sono uguali, p = 0 -> cout = a
cout <= a when p = '0' else cin;
sum <= p xor cin;

end MyFa;
```

A questo punto arrivati lo schematic corrispondente sarà:

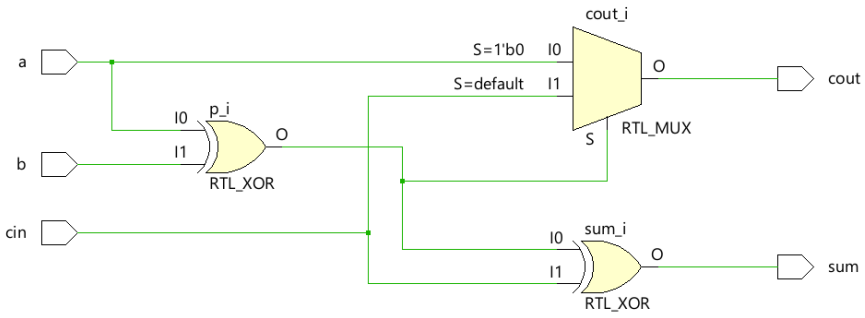


Figure 10: Schematic FA con segnale p e g

3.1.2 Vantaggi del mux

Il *MUX* riduce la complessità logica del carry-out, poiché andiamo a calcolare il carry-out mediante la selezione tra due valori invece di avere una serie di *AND* e *OR*. Un altro aspetto importante da tenere in considerazione è l'efficienza; usare il *MUX* può essere più efficiente a causa di un minor numero di porte utilizzate, e per questo è preferibile utilizzare questa versione se implementato in circuiti più complessi come il ripple carry e il carry select

3.2 Ripple Carry Adder

Andiamo ora a implementare il nostro RCA in VHDL, per procedere ci sono diverse modalità :

a. strutturale ; fa riferimento alla struttura, si avvale dunque dell'utilizzo di *component*, ci interfacciamo con i componenti del circuito.

b. behavioral : non facciamo riferimento ai componenti del circuito.

Di entrambe le modalità ne analizzeremo codice e *schematic*.

3.2.1 Rappresentazione gerarchica

Come precedentemente annunciato, questa utilizza componenti già creati, nel caso del RCA, da quello che abbiamo potuto constatare nei paragrafi precedenti, questo utilizza diversi Full Adder.

Dopo aver definito il componente full adder creiamo le due macro aree che ci servono:

a. entity

b. architecture

Scriviamo nel seguente modo:

```
entity ripple_carry is
    Port (cin : in bit;
           A : in bit_vector(7 downto 0);
           B :in bit_vector(7 downto 0);
           Sum : out bit_vector(7 downto 0);
           cout : out bit);
           --Cout è il bit più significativo Sum(8)

end ripple_carry;

architecture MyRc of ripple_carry is
    --dobbiamo specificare le componenti
    component FullAdder is
    Port(a, b, cin : in bit;
    cout, sum : out bit);
    end component;
    --metto i segnali intermedi
    signal Cmed : bit_vector(8 downto 0);
    --ora possiamo iniziare il begin
begin
    --facciamo il for per mappare il fa nel ripple carry
    MyFor : for i in 0 to 7 generate
        mappa : FullAdder port map (A(i), B(i),
        Cmed(i), Cmed(i + 1), Sum(i));
        end generate;
        Cmed(0) <= cin;
        cout <= Cmed(8);
        --abbiamo finito
end MyRc;
```

Partiamo dal presupposto di voler costruire un ripple carry a 8 bit, come prima cosa avrà bisogno di un segnale di ingresso che andiamo a chiamare c_{in}, che sarà rappresentato da un solo bit; gli altri ingressi invece sono dei vettori di bit, che andiamo a chiamare A e B e che avranno lunghezza 8, scrivere l'espressione (7 donwto 0) indica che il vettore è formato dai bit di A nel seguente modo:

A(7 downto 0) = A(7) A(6) A(5) ... A(0)

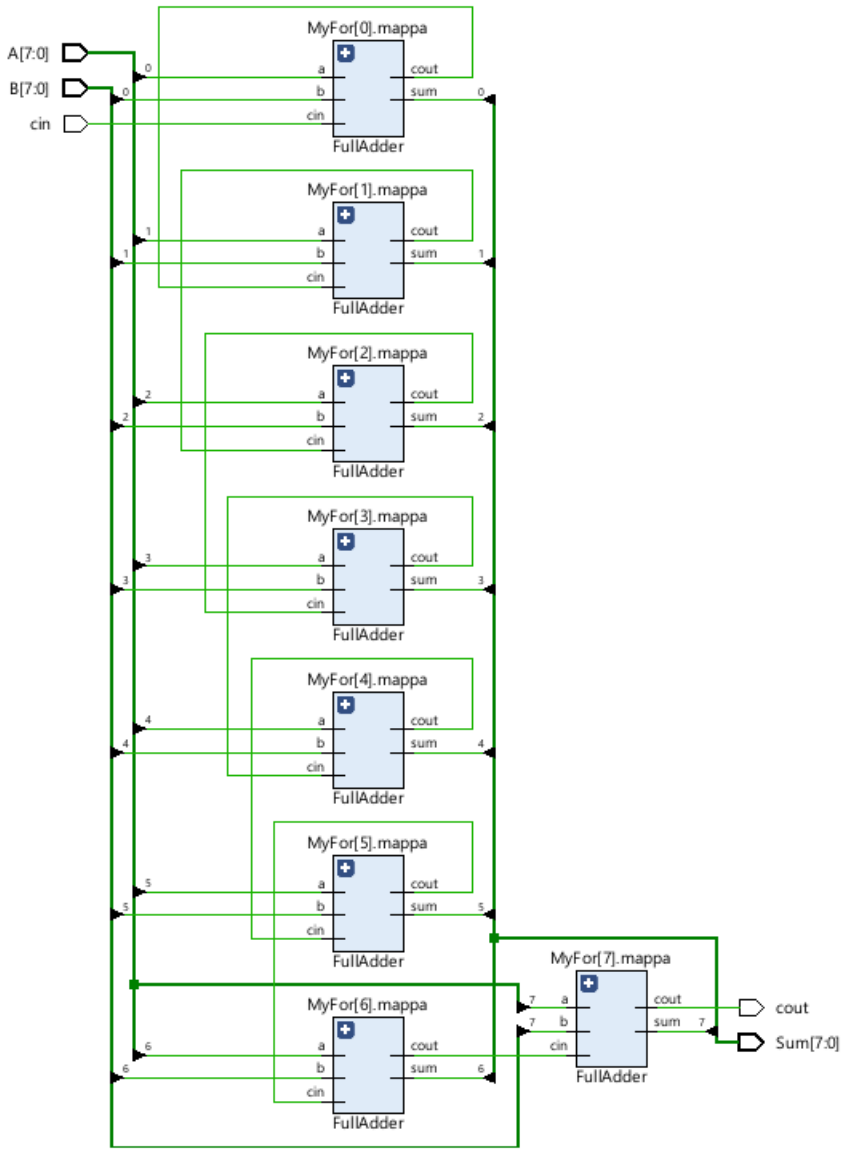
Avviene dunque quella che si chiama **indicizzazione decrescente**, esiste anche quella **crescente** che invece è del tipo:

$$A(0 \text{ upto } 7) = A(0)\ A(1)\ \ldots\ A(7)$$

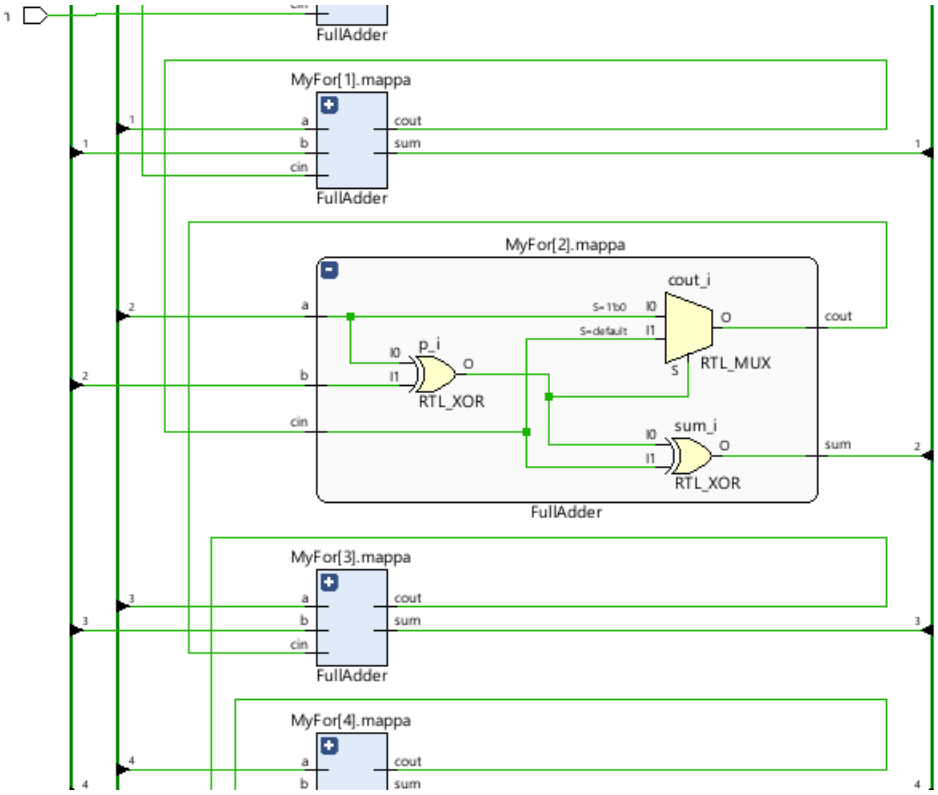
Si possono usare senza nessun problema entrambe, è importante sapere, però che se si sceglie di utilizzare una rappresentazione per un determinato circuito, poi anche gli altri vettori dovranno utilizzarla; questo perchè se scrivessimo qualcosa del tipo A and B, e i due vettori avessero rappresentazioni differenti avremmo per esempio A(0) and B(7) invece che A(0) and B(0) oppure A(7) and B(7).
Dopo di che passiamo agli output che avremo, che saranno un vettore somma da 8 bit e un Cout di un bit.
Si noti che all'interno della entity non sono specificati i segnali interni al circuito, che invece dovranno essere richiamati all'interno dell'architettura.
L'architettura data la natura gerarchica, prende il component Full Adder e all'interno dello *statement* lo richiamerà per le volte che gli serve(nel nostro specifico caso del rca, avremo un numero di full adder pari a quello di bit con cui lavora il ripple carry); come precedentemente annunciato, prima del *begin* ci serviranno i segnali intermedi che andremo a mettere tutti inisieme all'interno di un vettore di bit. Le Port invece dovranno avere stesso nome e stesso ordine di dichiarazione del component.
Ora possiamo iniziare la nostra procedura di "*mappatura*" attraverso un for e infine assegnamo i bit di uscita.
Finito tutto questo possiamo vedere cosa ne esce fuori graficamente, attraverso lo schematic.

3.2.2 Schematic gerarchico

Questo è lo schematic che viene fuori dal codice appena rappresentato:



Facciamo un paio di considerazioni anche sul grafico ottenuto, si può notare facilmente l'insieme di fulladder in successione, e in più cliccando su uno qualunque di questo avremmo anche la sua struttura:



Questo a riprova del fatto che la struttura creata è gerarchica.

3.2.3 Rappresentazione behavioral

Vediamo ora la versione behavioral ovvero quella comportamentale, che non si avvale dell'utilizzo del FA.

```
entity r_carry2 is
--  versione senza component
-- prendo in ingresso i vettori A e B composti da 8 bit e un bit di cin
-- in output invece avrò un vettore di 9 elementi(in generale n + 1 elementi)
Port(A, B : in bit_vector(7 downto 0);
     cin: in bit;
     S : out bit_vector(8 downto 0));

end r_carry2;

architecture My_second_carry of r_carry2 is
--in questo caso prendiamo i signal p e g
--p corrisponde allo xor tra A e B
--g corrisponde all'and tra A e B
signal p,g : bit_vector(8 downto 0);
```

```
signal c : bit_vector(9 downto 0);

--iniziamo
begin
--ricordiamo di fare le operazioni tra vettori della stessa dimensione
p(8) <= A(7) xor B(7);
g(8) <= A(7) and B(7);
p(7 downto 0) <= A xor B;
g(7 downto 0) <= A and B;

c(9 downto 1) <= g or (p and c(8 downto 0));
s <= p xor c(8 downto 0);

end My_second_carry;
```

Qui come si può notare, a differenza del precedente Ripple Carry rappresentato, nell'architettura andiamo a mettere i segnali di *propagazione*[1], e di *generate*[2]; procediamo quindi andando a calcolare p e g, ricordando che le operazioni le possiamo fare su vettori delle stesse dimensioni, e poi i due output.

3.2.4 Schematic comportamentale

Di seguito viene rappresentato lo schematic, che è ben diverso dal precedente:

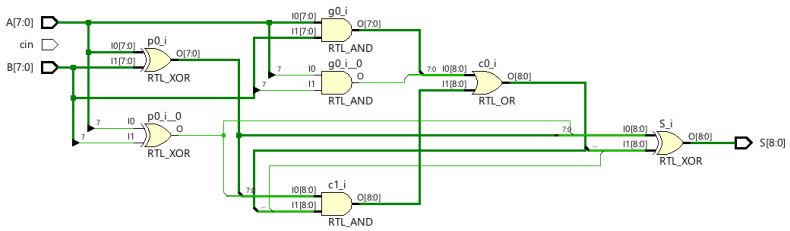


Figure 11: Schematic behavioral RCA

[1] questo segnale corrisponde allo xor degli ingressi
[2] corrisponde all'and degli ingressi

3.3 Carry Select Adder

Procediamo visualizzando l'implementazione del codice VHDL del *Carry Select Adder*.

```
entity carry_select is
    --mi dichiaro le porte che saranno i vettori A, B in
    --ingresso e poi S in uscita
    -- So ha dimensioni di 9 perchè incorpora anche l'ultimo bit
    Port(Acs,Bcs : in bit_vector(15 downto 0);
        So : out bit_vector(16 downto 0));

end carry_select;

architecture My_carry_select of carry_select is

--component mux
component mux is
Port(a, b, sel : in bit;
     s_out : out bit);
end component;

--componente full adder
component FullAdder is
    Port(a, b, cin : in bit;
    cout, sum : out bit);
end component;

--componente ripple carry
component ripple_carry is
    Port (cin : in bit;
           A : in bit_vector(7 downto 0);
           B :in bit_vector(7 downto 0);
           Sum : out bit_vector(7 downto 0);
           cout : out bit);
end component;

--metto i segn

--quelli intermedi
signal Cout0, Cout1, Cout2, Sout, Coverflow : bit ;
--quelli per la somma di 0 e 1
signal Som0, Som1 : bit_vector(7 downto 0);

begin
--ora dobbiamo mappare
```

```
RCA_primo_blocco : ripple_carry port map('0', Acs(7 downto 0),
Bcs(7 downto 0), So(7 downto 0),Cout0);
RCA0 : ripple_carry port map('0', Acs(15 downto 8),
Bcs(15 downto 8), Som0,Cout1);
RCA1 : ripple_carry port map('1', Acs(15 downto 8),
Bcs(15 downto 8), Som1,Cout2);

--gestiamo i mux attraverso il for

MyFor: for i in 0 to 7 generate
    Mux_iesimo : mux port map(Som0(i), Som1(i),  Cout0, So(8 + i));
end generate MyFor;

--Mux grande
Mux_dec: mux port map(Cout1, Cout2, Cout0, Sout);

--gestico l'overflow con un Full Adder
FA: fullAdder port map (Acs(15), Bcs(15), Sout,Coverflow ,So(16));

end My_carry_select;
```

3.3.1 Descrizione entity CSA

Possiamo osservare che, nella prima parte, l'**entity**, abbiamo dichiarato due vettori da 16 bit, *Acs* e *Bcs*, che costituiscono i due numeri da sommare e un ulteriore vettore, da 17 bit, che abbiamo chiamato *So*, in cui sarà presente la somma finale.

3.3.2 Descrizione architecture CSA

La seconda parte del codice VHDL è ampiamente occupata dalla descrizione dell'architecture del CSA. Iniziamo con il riportare le tre componenti che caratterizzano la struttura del Carry Select Adder, spiegate nei precedenti paragrafi, esse sono: il *multiplexer*, il *full adder* e il *ripple carry adder*.
Subito dopo aver aver dichiarato le componenti, descriviamo i segnali intermedi che verranno utilizzati all'interno del circuito:

- Cout0, Cout1, Cout2, Sout: sono utilizzati per i carry e il multiplexer
- Som0, Som1, sono i bit vector che calcolano la somma dei due ripple carry, uno che riceve un carry-in pari a 0 e l'altro pari a 1

Dopo la key words *begin* presentiamo i 3 blocchi di ripple carry, il primo si occupa di sommare i primi 8 bit meno significativi, che verrà salvata in So e l'ultimo bit in Cout0.
RCA0 e RCA1 servono per calcolare l'addizione dei rimanenti 8 bit più significativi RCA0 riceve cin pari a 0, RCA1 riceve cin uguale a 1. L'esito delle somme viene memorizzato nei due vettori precedentemente citati, ovvero Som0 e Som1. Il carry finale è salvato in Cout1 e Cout2, input dei mux utilizzati successivamente.

Proseguiamo poi con un ciclo for sui multiplexer che serve per selezionare, per ogni bit, il valore corretto, che può essere quello di Som0 se il segnale di selezione Cout0 è uguale a 0 oppure Som1 se Cout è pari a 1. Quindi ciascun mux seleziona tra Som0(i) e Som1(i) e il risultato viene riportato in So.
Il risultato finale viene memorizzato in Sout dopo che un ultimo multiplexer si occupa di scegliere tra Cout1 e Cout2 (in base al valore di Cout0).
Infine, nel caso in cui si presenti un overflow, viene utilizzato un full adder che gestisce eventuali riporti.

3.4 Schematic gerarchico CSA

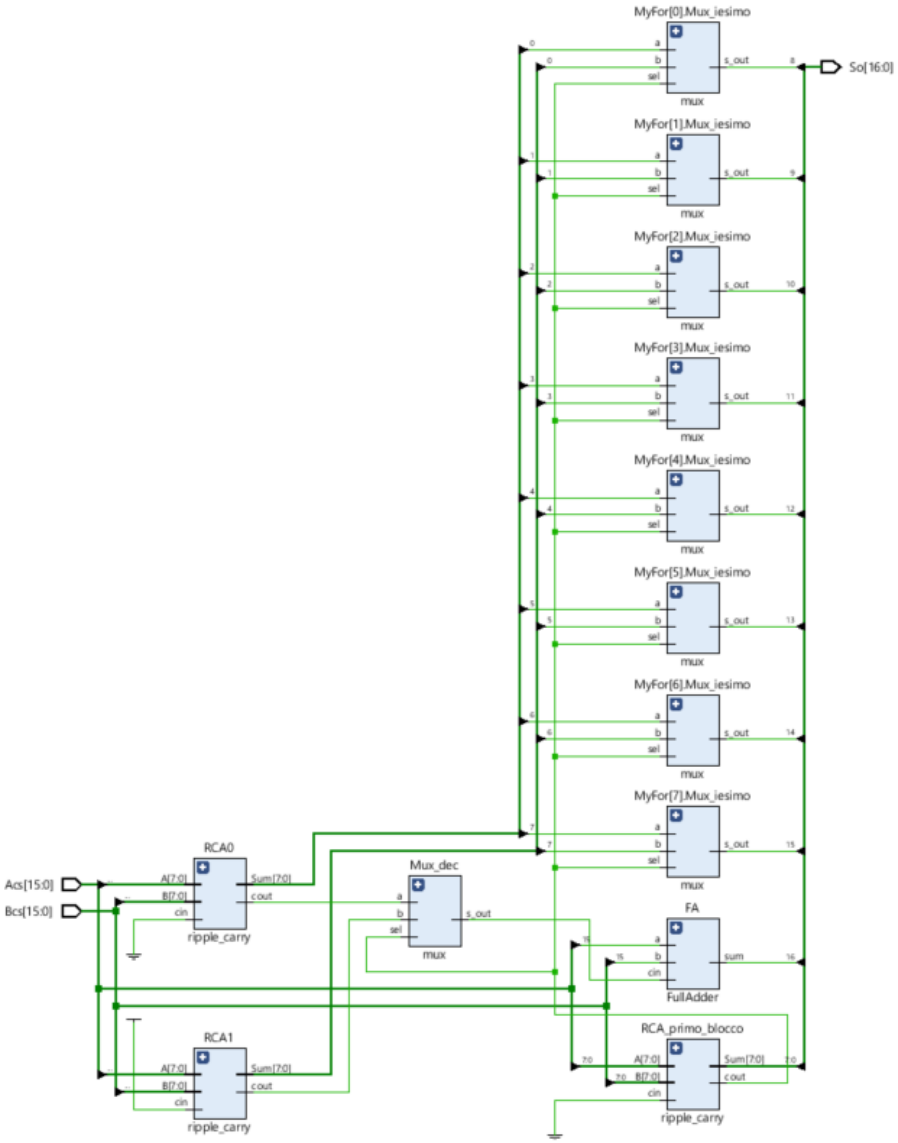


Figure 12: CSA con gestione dell'overflow

4 Simulazione e risultati

A questo punto, dopo aver scritto e visualizzato il nostro **carry select adder**, non rimane altro che andarlo a testare, questa operazione viene fatta attraverso quello che prende il nome di **test bench** che serve a simulare il funzionamento del circuito appena progettato.
Per avviare un file di simulazione, andiamo ad aggiungere una *source*, precisamente di simulazione, e poi andiamo a scrivere il codice.
Nel test bench, è fondamentale includere l'entità senza alcuna logica interna.

```
entity tb_Csa is
-- l'entity nel tb è vuota
end tb_Csa;

architecture My_test_bench of tb_Csa is

    component carry_select is
    Port(Acs,Bcs : in bit_vector(15 downto 0);
        So : out bit_vector(16 downto 0));

    end component;

    --metto i segnali

    signal IA, IB : bit_vector(15 downto 0);
    signal Somma : bit_vector(16 downto 0);

begin

    IA <= "0000000000000001";
    IB <= "0000000000000001";

    CSA : carry_select port map(IA, IB, Somma);

end My_test_bench;
```

In questo caso quello che andiamo a fare è dare come componente del test bench il circuito che vogliamo analizzare, e dichiarare i segnali di ingresso, ovvero i due vettori di bit e poi quello di uscita, sempre vettore di bit, che rappresenterà la somma.
Nel begin assegniamo semplicemente due numeri a IA e IB, nel nostro caso per iniziare ci siamo messi in una casistica molto semplice : $1 + 1 = 2$, e diamo i "dati" al CSA; si noti bene che in questa caso non vi è variazione di bit nel corso del tempo.
Abbiamo poi anche un ulteriore test, questa volta

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity tb_Csa is
-- l'entity nel tb è vuota
end tb_Csa;

architecture My_test_bench of tb_Csa is

    component carry_select is
    Port(Acs,Bcs : in bit_vector(15 downto 0);
        So : out bit_vector(16 downto 0));

    end component;

    --metto i segnali

    signal IA, IB : bit_vector(15 downto 0);
    signal Somma : bit_vector(16 downto 0);

begin
 CSA : carry_select port map(IA, IB, Somma);

    process
    begin
        IA <= "0000000000000000";
        wait for 20 ns;
        IA <= "0010101000101010";
        wait for 30 ns;
        IA <= "0000000011111111";
end process;

    process begin
        IB <= "0000000000000000";
        wait for 20 ns;
        IB <= "0010101000101010";
        wait for 30 ns;
        IB <= "0000000011111111";
end process;
end My_test_bench;
```

Dalla figura 13 e 14 è possibile notare graficamente, come il circuito gestisce l'input e l'output in funzione del tempo

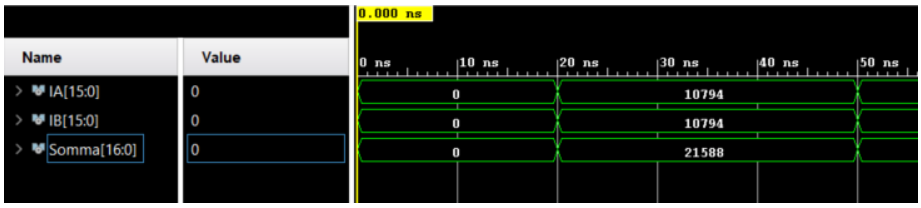


Figure 13: Simulazione con t = 0 ns

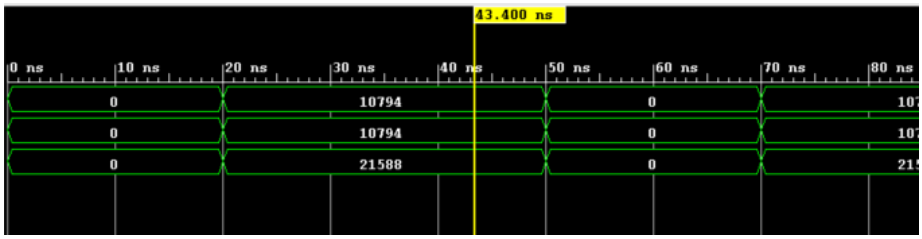


Figure 14: Simulazione con t = 43 ns

È possibile osservare che il circuito gestisce correttamente l'input calcolando correttamente il risultato.

5 Conclusioni finali

In sintesi, il progetto del **Carry Select Adder (CSA)** ha evidenziato l'efficacia di questa architettura rispetto ad altri circuiti sommatori, dimostrando come la somma parallela dei blocchi riduca significativamente i tempi di latenza.