



# Progetto di Elettronica Digitale

---

Giorgio Ubbriaco  
209899



UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI  
INGEGNERIA INFORMATICA,  
MODELLISTICA, ELETTRONICA  
E SISTEMISTICA

DIMES

---

# Indice

<b>Introduzione</b>	<b>3</b>
<b>MyDefinitions</b>	<b>3</b>
<b>FA</b>	<b>4</b>
<b>RTL Analysis – Schematic</b>	<b>6</b>
<b>CSA</b>	<b>7</b>
<b>Descrizione CSA</b>	<b>7</b>
<b>RTL Analysis – Schematic</b>	<b>12</b>
<b>Synthesis – Schematic</b>	<b>15</b>
<b>Implementation – Device</b>	<b>16</b>
<b>SimCSA</b>	<b>20</b>
<b>Behavioral Simulation</b>	<b>22</b>
<b>Post-Synthesis Timing Simulation</b>	<b>23</b>
<b>Post-Implementation Timing Simulation</b>	<b>24</b>
<b>Power On-Chip</b>	<b>25</b>

# Introduzione

Progettare un Carry Select Adder generico ad n bit e con caratteristiche reali a 16 bit considerando operandi in complemento a due. Svolgere il progetto tramite l'utilizzo del software Vivado ed utilizzando il linguaggio descrittivo VHDL. Effettuare la sintesi e l'implementazione del circuito logico ottenuto e le corrispondenti simulazioni Post-Synthesis Timing Simulation e Post-Implementation Timing Simulation. Commentare, inoltre, i report ottenuti.

# MyDefinitions

Il package MyDefinitions contiene le definizioni delle due costanti utilizzate all'interno della progettazione del Carry Select Adder: `n` è il numero di bit di cui sono composti gli addendi da sommare ed `nFA` è il numero di Full Adder che conterrà ogni blocco del CSA, cioè rispettivamente  $\frac{n}{2}$  ognuno. Tali costanti permetteranno, quindi, di mantenere all'interno del codice una genericità per quanto riguarda il numero di bit da sommare. In questo caso, essendo che le specifiche richieste sono quelle di considerare un circuito CSA con caratteristiche reali a 16 bit, imposterò il valore di `n` e di `nFA` rispettivamente pari a 16 e 8.

The screenshot shows a VHDL code editor with the following content:

```
MyDef.vhd
D:/Vivado Projects/Progetto1/Progetto1.srcs/sources_1/new/MyDef.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

package MyDefinitions is
    constant n: integer:=16; -- numero di bit da sommare
    constant nFA: integer:= n/2; -- numero di Full-Adder per blocco
end package;
```

*Figura 1 MyDef*

# FA

Il Full Adder è un circuito logico caratterizzato da tre ingressi e due uscite. La sua funzionalità è quella di eseguire una somma tra due numeri espressi in formato binario con lunghezza di parola a un bit. In logica binaria, il FA esegue la seguente operazione:

$$A + B + Cin = S + Cout$$

dove **A** è il primo addendo, **B** è il secondo addendo, **Cin** è il riporto in “entrata”, **S** è il bit meno significativo della somma in uscita ed, infine, **Cout** è il bit più significativo della somma ottenuta, anche conosciuto come riporto in “uscita”. Ad ogni variabile corrisponde un bit (0 oppure 1). Pertanto, in ingresso sono inseriti i due bit da sommare e l’eventuale bit di riporto, mentre in uscita vengono forniti la somma e il riporto.

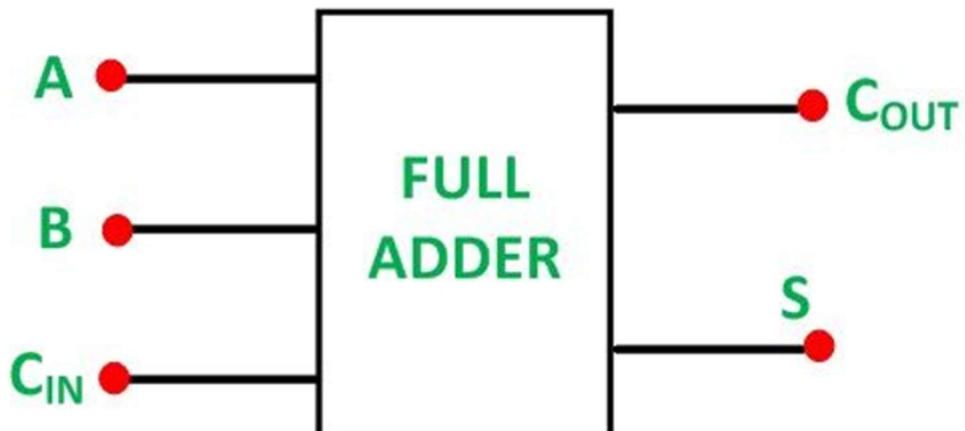


Figura 2 Full Adder generico

Il corrispondente codice descrittivo del componente logico Full Adder sarà:

FA.vhd

D:/Vivado Projects/Progetto1/Progetto1.srcs/sources\_1/new/FA.vhd

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity FA is
5     port(
6         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    );
12 end FA;
13
14
15 architecture Behavioral of FA is
16
17 begin
18     S <= A xor B xor Cin;
19     Cout <= (A and B) or ( (A xor B) and Cin );
20
21 end Behavioral;
22
```

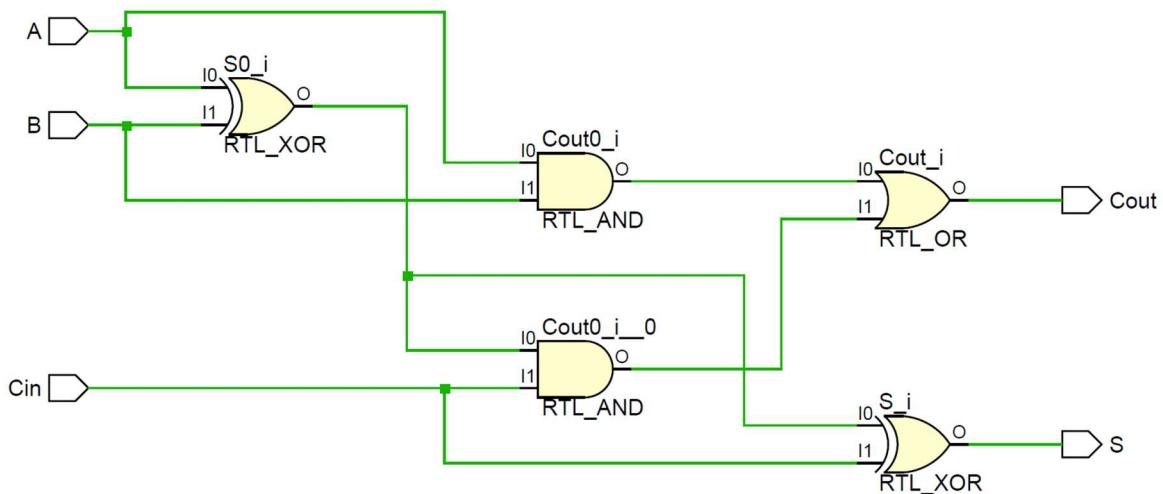
*Figura 3 FA*

Un’*entity declaration* è un’astrazione di un dispositivo che rappresenta un componente o anche un sistema completo. Una dichiarazione di *entity* descrive l’I/O di un componente. In questo caso trattandosi di un Full Adder vengono specificati tre ingressi ( $A$ ,  $B$  e  $Cin$ ) e due uscite ( $S$  e  $Cout$ ). Inoltre, viene definita un’architettura che permette di descrivere il *design* del componente, cioè cosa effettivamente è presente all’interno del blocco. Un’architettura può contenere *statements* sia concorrenti che sequenziali in base all’uso. In questo caso viene definito il *design* del Full Adder il quale calcolerà il bit di somma  $S$  facendo la  $\oplus$  a tre ingressi ( $A$ ,  $B$  e  $Cin$ ) e il bit di riporto in uscita  $Cout$  come  $(A \cdot B) + ((A \oplus B) \cdot Cin)$ .

I segnali I/O sono stati definiti tramite lo *standard IEEE 1164*, conosciuto anche come *std\_logic*. Tale *standard* permette di avere più stati oltre a quelli definiti dal tipo bit (0 e 1) come, ad esempio, 'U' che sta a significare che il segnale non ha un valore di inizializzazione oppure 'X' che sta a significare che il segnale non sta assumendo nessun valore.

## RTL Analysis – Schematic

Effettuando l'RTL Analysis tramite il comando Open Elaborated Design otteniamo lo **schematic** corrispondente al Full Adder. Lo schematic è una rappresentazione grafica della **netlist**, che viene usata nel campo della progettazione elettronica per indicare l'insieme delle connessioni (*net*) elettriche di un circuito elettronico. Se la netlist contiene anche altre informazioni più approfondite, come in questo caso, viene generalmente considerata come un **Hardware Description Language** (HDL) come ad esempio **VHDL** e **Verilog**, cioè un linguaggio specifico per fornire i dati in ingresso a programmi di simulazione circuitale. Le netlist possono essere fisiche o logiche, basate sulle istanze o sulle connessioni. Le **netlist net-based** descrivono, ad esempio, tutte le istanze e i loro attributi, cioè elencano ciascuna net e le porte a cui sono connesse.



Quindi, attraverso lo *schematic* possiamo rivedere le porte, le gerarchie e le connettività, cioè possiamo capire cosa sta succedendo all'interno del design e come lo *strumento* ha interpretato il codice FA. Infatti, possiamo notare che i segnali *A* e *B* sono mandati in input alla prima *XOR* in alto a destra e il corrispondente output mandato in input all'ultima *XOR* in basso a sinistra insieme a *Cin* tale che l'output ottenuto corrisponderà ad *S*. Per quanto riguarda l'output *Cout*, esso sarà ottenuto tramite l'*OR* tra l'output dell'*AND* tra *A* e *B*, e l'*AND* tra la *XOR* in alto a sinistra e *Cin*.

# CSA

## Descrizione CSA

Il Carry Select Adder (conosciuto anche con l'acronimo **CSA**) è un addizionatore, cioè un circuito logico atto ad eseguire l'operazione aritmetica di addizione tra due numeri espressi in forma binaria. Rispetto al *RCA* (acronimo di Ripple Carry Adder), composto da una cascata di *FA*, il *CSA* risulta essere più veloce introducendo un certo grado di ridondanza all'interno del circuito. Infatti, considerando una somma ad  $n$  bit tra due ipotetici operandi  $A$  e  $B$ , ciascuno di  $n$  bit, possiamo considerare blocchi di *FA* da  $\frac{n}{2}$  bit ciascuno. Un primo blocco calcolerà la somma tra i primi  $\frac{n}{2}$  bit di  $A$  e i primi  $\frac{n}{2}$  bit di  $B$  ed, infine, due blocchi di *FA* in parallelo, l'uno con il segnale  $Cin = 0$  e l'altro con  $Cin = 1$ , calcoleranno la somma degli  $\frac{n}{2}$  bit restanti. Infine, ogni risultato ottenuto dai blocchi in parallelo verrà inviato in input ad un *MUX* (acronimo di Multiplexer) che, tramite il bit selettori *Cout* del primo blocco di *FA* iniziale, deciderà quale risultato tra i blocchi in parallelo dovrà essere processato come somma di output. Proprio questo parallelismo permette di velocizzare il calcolo. Chiaramente, questo vantaggio comporta un prezzo da pagare in termini di porte logiche usate.

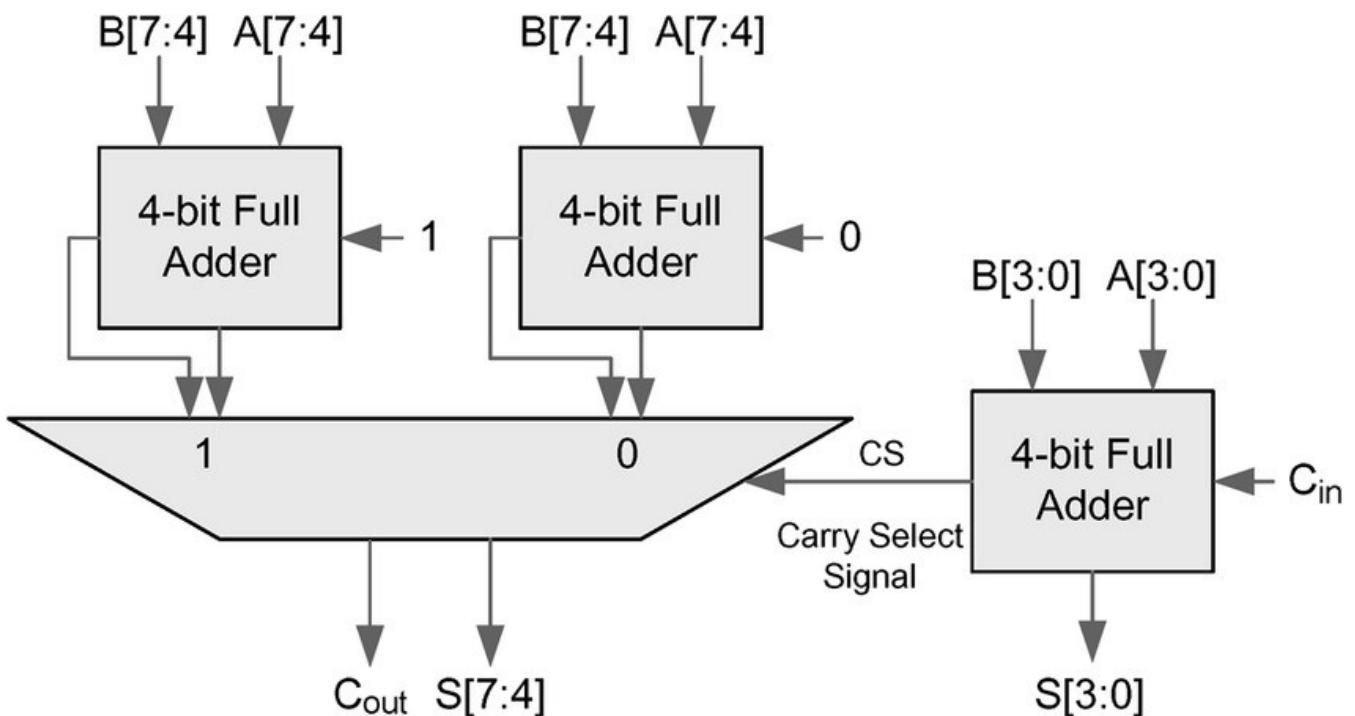
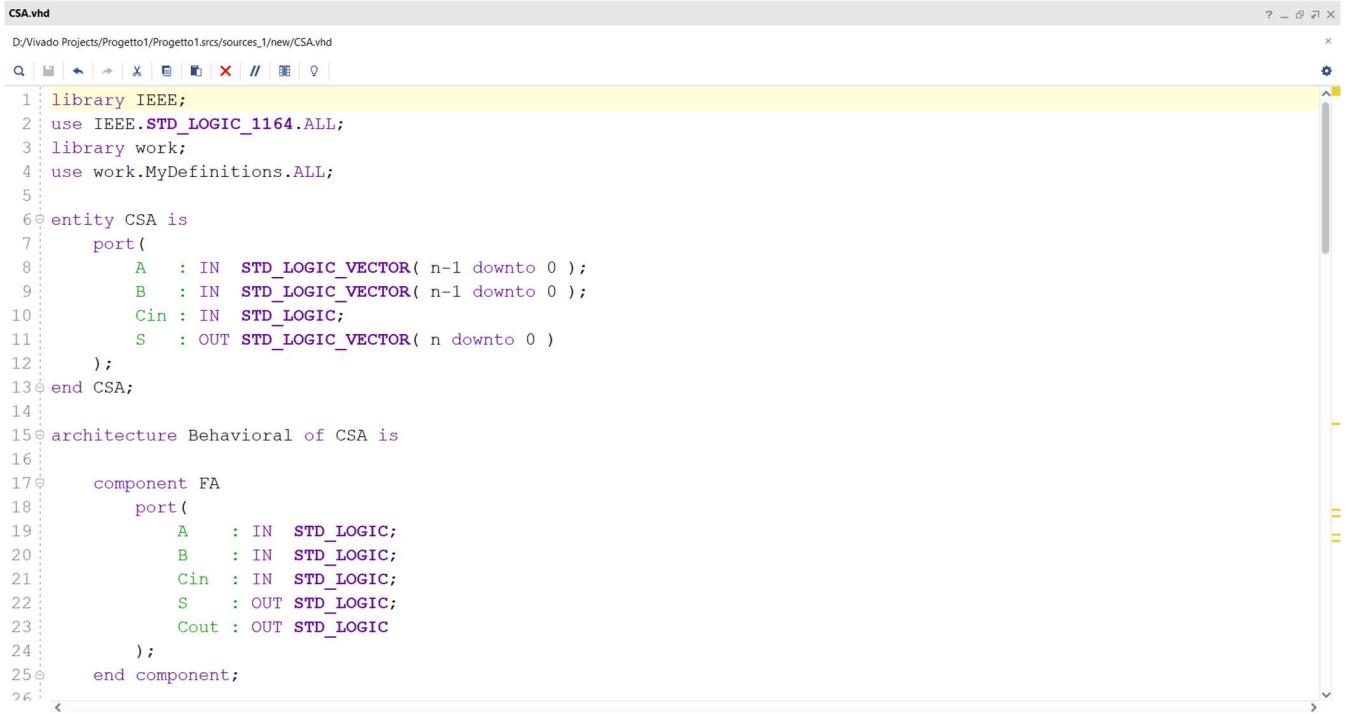


Figura 4 Carry Select Adder ad 8 bit

Il corrispondente codice descrittivo del componente logico Carry Select Adder sarà:



The screenshot shows a VHDL code editor window titled "CSA.vhd". The code is as follows:

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

entity CSA is
    port(
        A : IN STD_LOGIC_VECTOR( n-1 downto 0 );
        B : IN STD_LOGIC_VECTOR( n-1 downto 0 );
        Cin : IN STD_LOGIC;
        S : OUT STD_LOGIC_VECTOR( n downto 0 )
    );
end CSA;

architecture Behavioral of CSA is

component FA
    port(
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        Cin : IN STD_LOGIC;
        S : OUT STD_LOGIC;
        Cout : OUT STD_LOGIC
    );
end component;
```

Ho dichiarato l'entity *CSA* avente  $n$  bit<sup>1</sup> in ingresso sia per l'input *A* che per l'input *B*, l'ingresso *Cin* e gli  $n+1$  bit in uscita per l'output *S* che comprenderà, ovviamente, anche il riporto in uscita *Cout*. Inoltre, nell'architecture di *CSA*, ho instanziato il component Full Adder affinché esso possa essere implementato nel circuito generale del *CSA*.

<sup>1</sup> Si può specificare la genericità di  $n$  perché è stato importato il package *MyDefinitions* precedentemente descritto

```
CSA.vhd  
D:/Vivado Projects/Progetto1/Progetto1.srs/sources_1/new/CSA.vhd  
  
26  
27 signal carry_iniziale: STD_LOGIC_VECTOR( nFA downto 0 );  
28  
29 signal carry_z: STD_LOGIC_VECTOR( nFA+1 downto 0 );  
30 signal zero : STD_LOGIC;  
31  
32 signal carry_u: STD_LOGIC_VECTOR( nFA+1 downto 0 );  
33 signal uno : STD_LOGIC;  
34  
35 signal selettore : STD_LOGIC; -- selettore che deciderà l'uscita dei MUX  
36  
37 signal Ing1 : STD_LOGIC_VECTOR( nFA downto 0 ); -- vettore ingressi (corrispondenti alle uscite di FA_u) dei MUX  
38 signal Ing0 : STD_LOGIC_VECTOR( nFA downto 0 ); -- vettore ingressi (corrispondenti alle uscite di FA_z) dei MUX  
39 signal OutM : STD_LOGIC_VECTOR( nFA downto 0 ); -- vettore uscite dei MUX  
40  
41
```

Successivamente, ho dichiarato vari *signal* tra i quali troviamo *carry\_iniziale* che corrisponde al *trasporto del riporto* tra i vari Full Adder che compongono il blocco iniziale, *carry\_z* e *carry\_u* che corrispondono rispettivamente al *trasporto del riporto* dei blocchi in parallelo con input *Cin* = 0 e *Cin* = 1 e infine *zero* ed *uno* che corrispondono rispettivamente a *Cin* = 0 e a *Cin* = 1 per i blocchi che lavorano in parallelo. Il *signal seletto* corrisponde al bit che permetterà ai vari *MUX* (essendo che tale *signal* viene *prolungato* ad ogni *MUX* implementato nel *CSA*) di decidere quale ingresso in essi potrà diventare il loro output. Inoltre, ho implementato altri tre *signal*, utili sempre ai multiplexer, tra cui: *Ing1* che corrisponde ad uno degli input dei *MUX*, per la precisione l'uscita i-esima del Full Adder i-esimo del blocco in parallelo con *Cin* = 1. Analogamente per *Ing0* per i Full Adder con *Cin* = 0. Infine, il *signal OutM* è il vettore che conterrà gli output di tutti i *MUX* implementati nel *CSA*. Ovviamente, *carry\_z* e *carry\_u* avranno  $nFA+1$  bit<sup>2</sup> poiché il bit in più servirà a gestire il complemento a due dei blocchi in parallelo che gestiscono infatti i bit più significativi del risultato.

<sup>2</sup> Ricordando che il numero di Full Adder per blocco  $nFA = \frac{n}{2}$

CSA.vhd

D:\Vivado Projects\Progetto1\Progetto1.srs\sources\_1\new\CSA.vhd

```
begin
    -- implemento il blocco di FA iniziale che somma i primi nFA (=> n/2) bit
    for_iniziale: for i in 0 to nFA-1 generate
        FA_iniziale: FA port map( A(i), B(i), carry_iniziale(i), S(i), carry_iniziale(i+1) );
    end generate for_iniziale;

    carry_iniziale(0) <= Cin;
    selettore <= carry_iniziale( nFA ); -- il selettore dei MUX sarà il Cout del blocco di FA iniziale

    zero <= '0';
    carry_z(0) <= zero; -- assegno '0' al Cin del primo blocco in parallelo
    uno <= '1';
    carry_u(0) <= uno; -- assegno '1' al Cin del secondo blocco in parallelo

    -- implemento i blocchi in parallelo FA_z e FA_u rispettivamente con Cin = zero e Cin = uno
    for_FAzu: for j in 0 to nFA generate
        ife: if j=nFA generate
            FA_Mz: FA port map( A(n-1), B(n-1), carry_z(nFA), Ing0(nFA), carry_z(nFA+1) );
            FA_Mu: FA port map( A(n-1), B(n-1), carry_u(nFA), Ing1(nFA), carry_u(nFA+1) );
        end generate;
        ifl: if j<nFA generate
            FA_Lz: FA port map( A(j+nFA), B(j+nFA), carry_z(j), Ing0(j), carry_z(j+1) );
            FA_Lu: FA port map( A(j+nFA), B(j+nFA), carry_u(j), Ing1(j), carry_u(j+1) );
        end generate;
    end generate;
end generate;
```

Inizializzo un for *for\_iniziale* dove instanzio i Full Adder del blocco iniziale e assegno i signal *carry\_iniziale(0)*, *selettore*, *carry\_z* e *carry\_u* ai corrispondenti valori. Successivamente inizializzo un secondo for *for\_FAzu* dove instanzio i blocchi in parallelo di Full Adder (rispettivamente con *Cin* = 0 e *Cin* = 1) gestendo ovviamente il complemento a due.

```

CSA.vhd
D:/Vivado Projects/Progetto1/Progetto1.srcs/sources_1/new/CSA.vhd
Q | F | A | D | X | B | I | C | // | E | G | ? | X |
64      FA_Lz: FA port map( A(j+nFA), B(j+nFA), carry_z(j), Ing0(j), carry_z(j+1) );
65      FA_Lu: FA port map( A(j+nFA), B(j+nFA), carry_u(j), Ing1(j), carry_u(j+1) );
66    end generate;
67 end generate;
68
69
70
71
72 -- implemento i MUX
73 for_mux: for w in 0 to nFA generate
74   -- ricordando che Ing1 è il risultato del w-esimo FA del blocco in parallelo
75   -- con Cin = '1' e Ing0 è il risultato del w-esimo FA del blocco in parallelo
76   -- con Cin = '0'
77   OutM(w) <= Ing0(w) when selettore = zero else
78     Ing1(w) when selettore = uno else
79       'X';
80   S(w+nFA) <= OutM(w);
81   -- l'indice w+nFA è pari all'indice del w-esimo MUX addizionato al numero di FA
82   -- così da ottenere l'indice corretto per salvare l'output del MUX all'interno
83   -- del vettore somma S
84 end generate for_mux;
85
86
87 end Behavioral;
88

```

Infine, inizializzo un terzo *for* *for\_mux* nel quale instanzio i *MUX* che serviranno a decidere quale dei suoi due ingressi, attraverso il selettore, sarà il suo output tale da far parte del vettore *S* di somma finale. Infatti, tramite il costrutto *when-else* e il *selettore*, viene deciso quale tra il *signal* *w*-esimo *Ing0* e *Ing1* sarà il *w*-esimo *OutM*. Infine, *OutM* viene assegnato al *w-simo+nFA* di <sup>3</sup> *S*.

---

<sup>3</sup> Utilizzo questo indice *w*-esimo+*nFA* proprio per riferirmi al corretto indice di somma finale del vettore *S*. Infatti, con *w* mi riferisco al *w*-esimo output del *MUX* in riferimento ai *w*-esimi Full Adder in parallelo e con *nFA* ottengo, pertanto, il corretto indice nel vettore finale di somma.

## RTL Analysis – Schematic

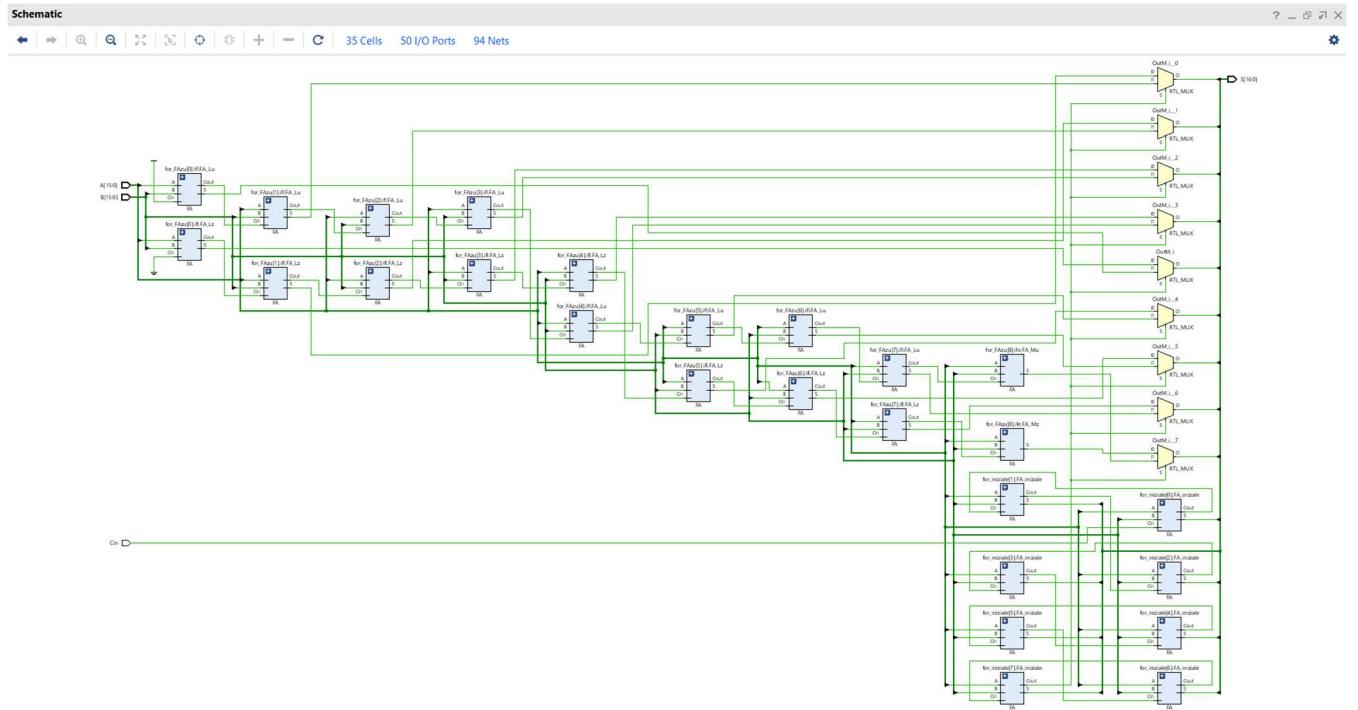
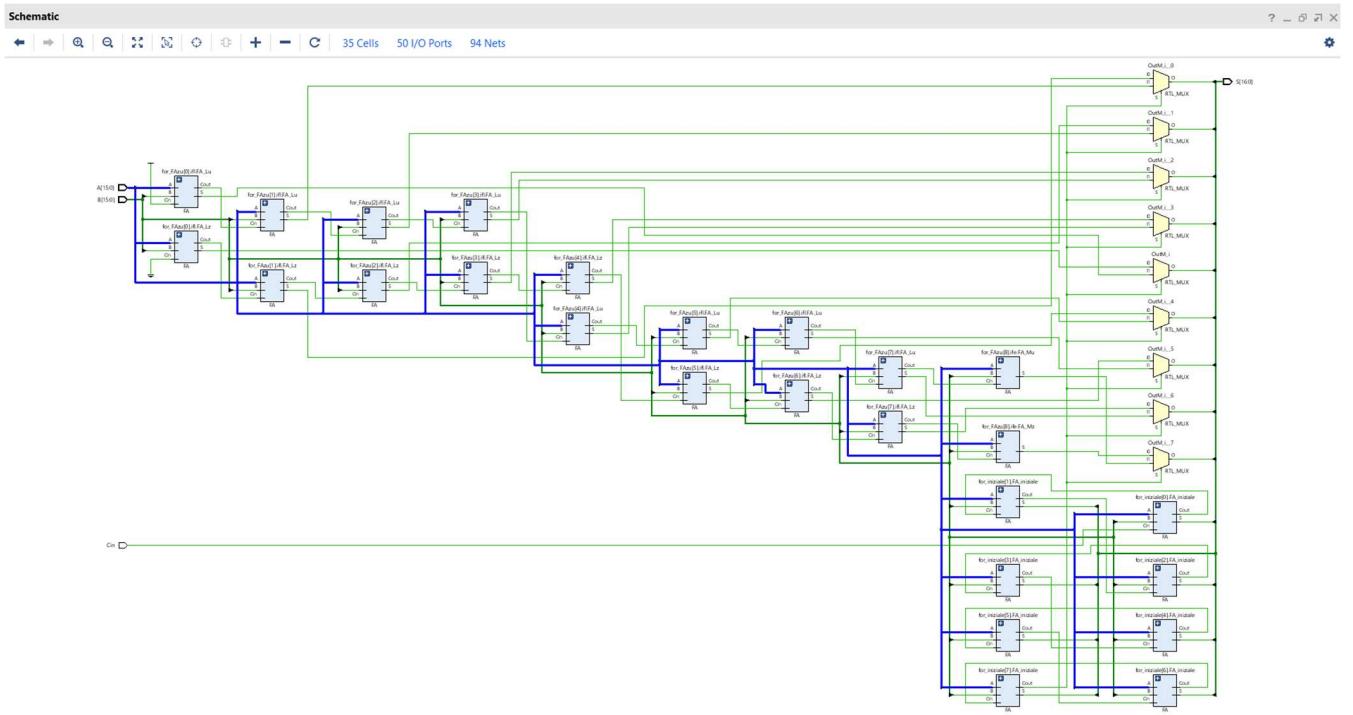
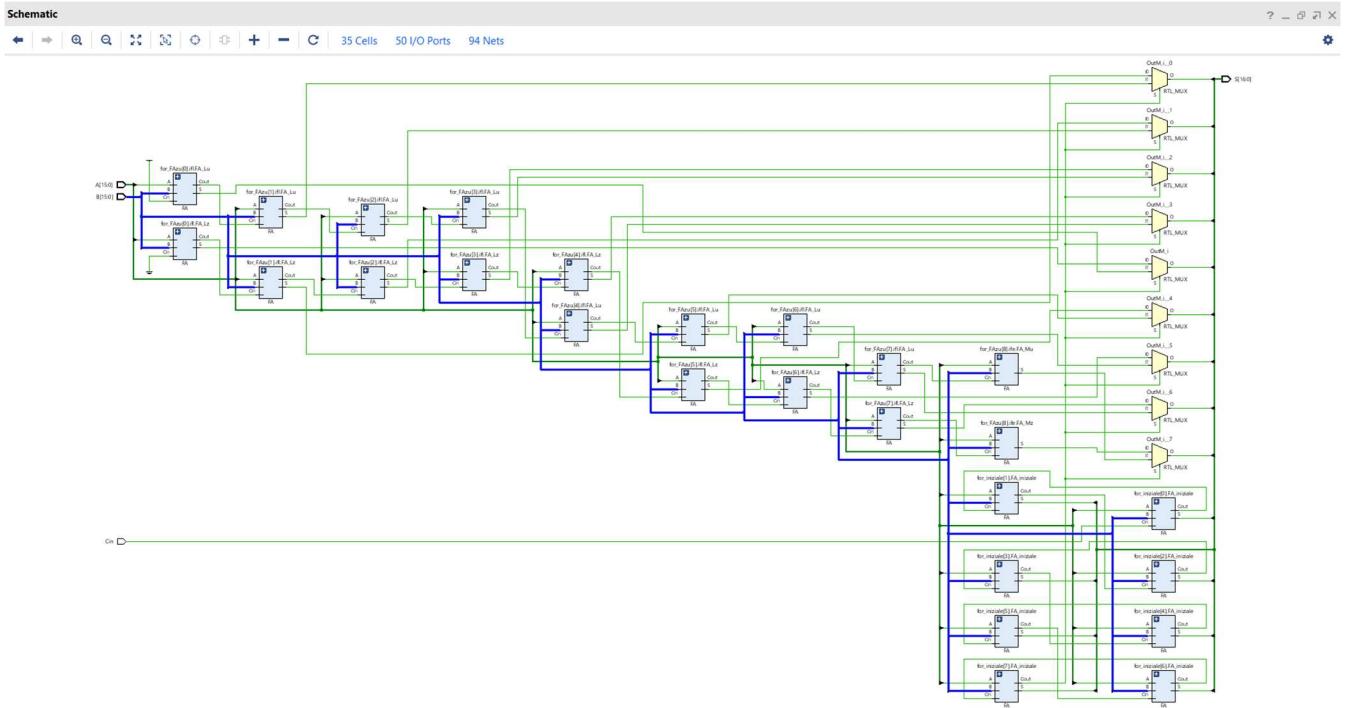


Figura 5 RTL Analysis - Schematic

Nel corrispondente schematic del Carry Select Adder possiamo notare come siano stati implementati in parallelo i Full Adder con rispettivamente  $Cin = 0$  e  $Cin = 1$  tale che ogni output viene mandato in input al corrispondente multiplexer. Nella parte in basso a destra dello schematic troviamo i Full Adder che compongono il blocco iniziale del circuito tale che ogni output, ovviamente, sarà direttamente assegnato al corrispondente indice nel vettore somma  $S$  (che nel caso dello schematic corrisponde al *Bus Net S*). Ovviamente, ogni FA riceverà i corrispondenti input dal *Bus Net A* e dal *Bus Net B*.

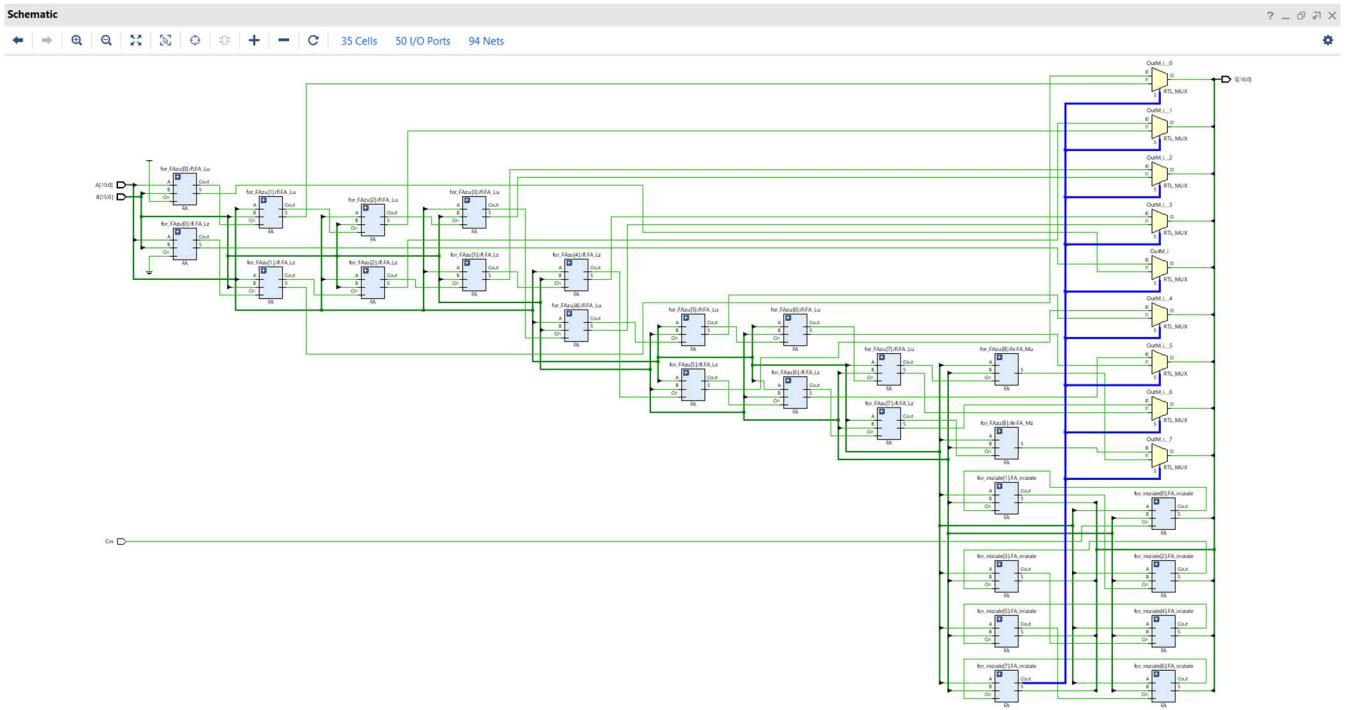


**Figura 6 RTL Analysis - Schematic highlighting Bus Net A**

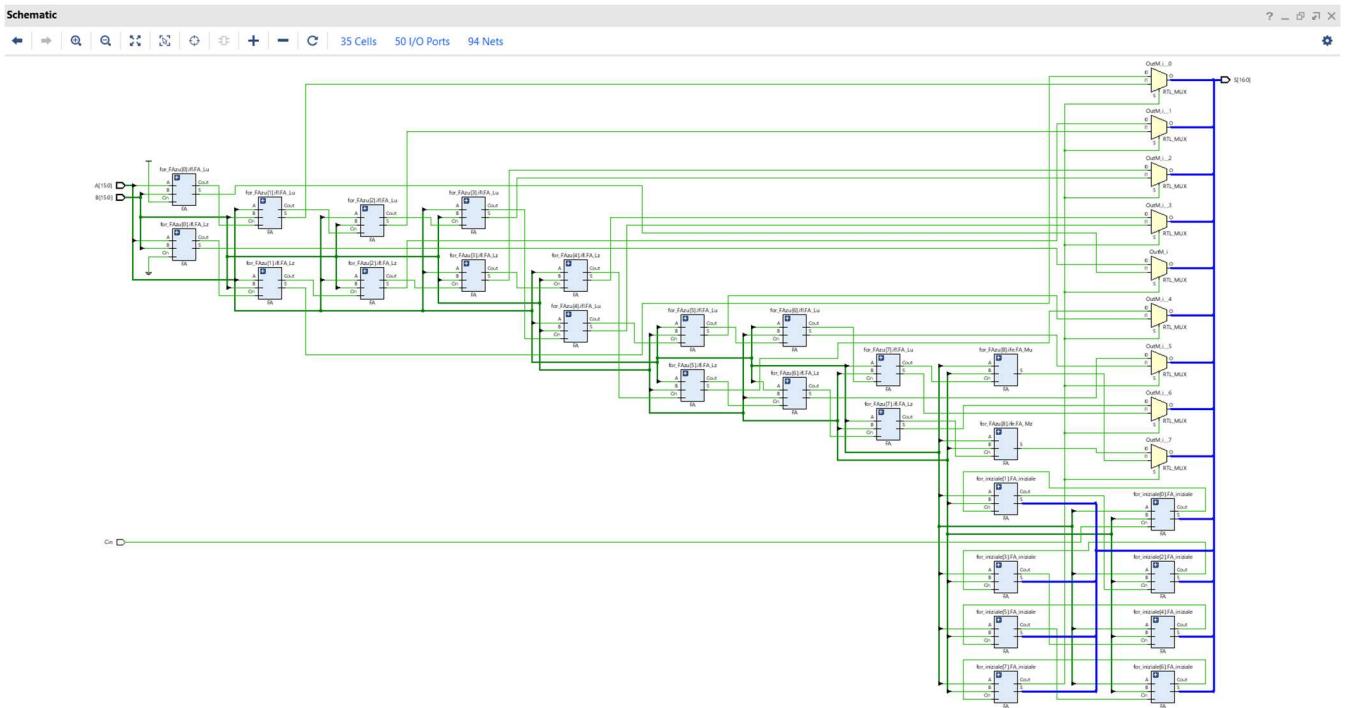


**Figura 7 RTL Analysis - Schematic highlighting Bus Net B**

Nella [figura 6](#) e nella [figura 7](#) ho evidenziato come gli input A e B sono mandati in ingresso ai rispettivi Full Adder.



**Figura 8 RTL Analysis - Schematic highlighting selettore**

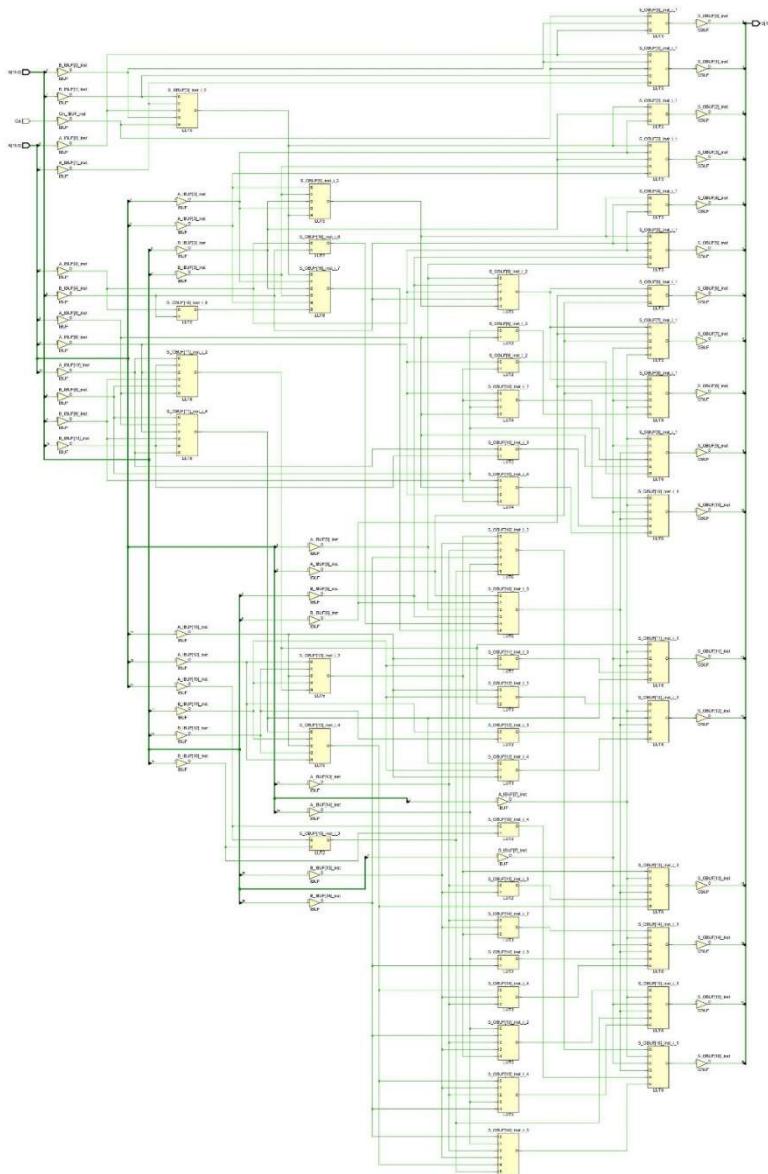


**Figura 9 RTL Analysis - Schematic highlighting Bus Net S**

Nella figura 8 e nella figura 9 ho evidenziato rispettivamente il selettore in input ad ogni multiplexer ed il Bus Net S.

## Synthesis – Schematic

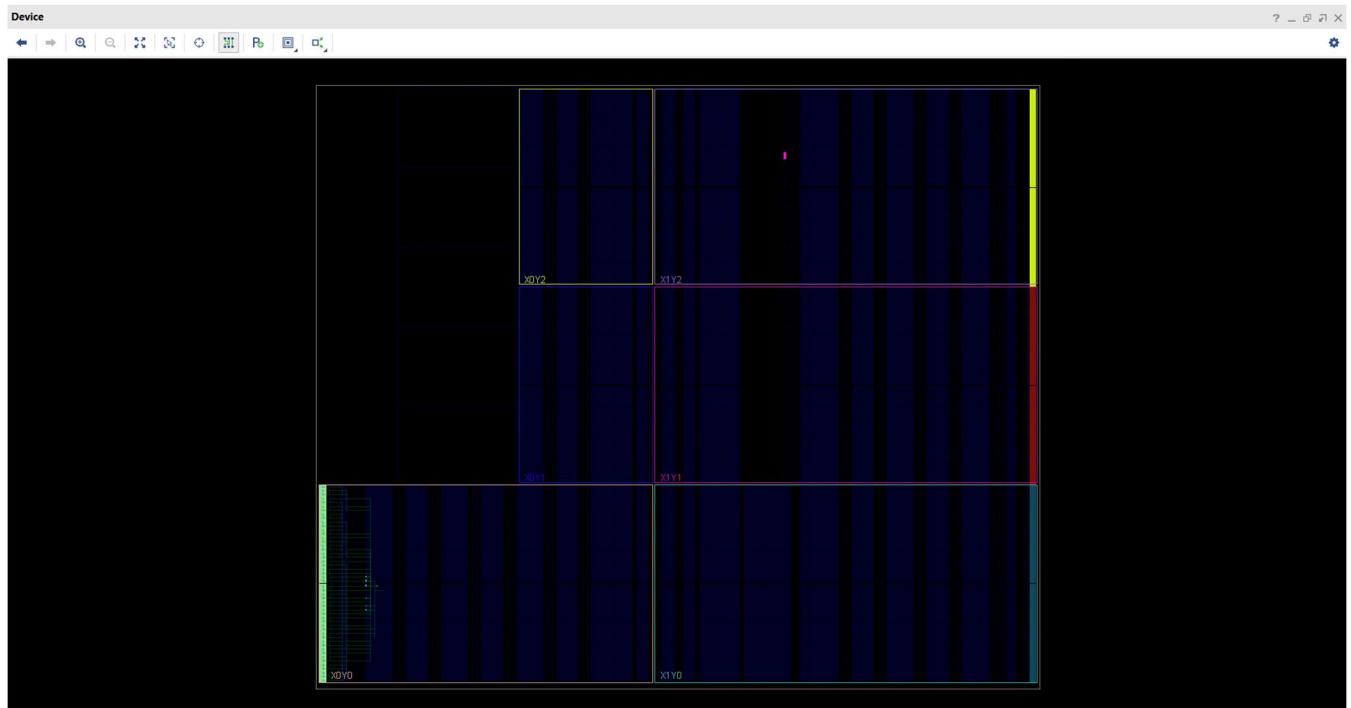
Una volta terminata l'analisi dell' RTL Analysis - Schematic, posso sintetizzare il circuito tramite *Run Synthesis* ottenendo di conseguenza lo schematic corrispondente al circuito sintetizzato. Posso notare la presenza degli *IBUF*, delle *LUTs* e degli *OBUF*. Gli *IBUF* sono *input buffer*, impostati dal tool per i *pin* di input del modulo progettato. Gli *OBUF* servono a *pilotare* i segnali dai moduli interni del circuito ai *pin* di output del modulo progettato. Infine, le *LUTs* sono delle strutture che permettono di associare ad ogni ammissibile combinazione di dati in ingresso una corrispondente configurazione dei dati in uscita.



### **Figura 10 Synthesis - Schematic**

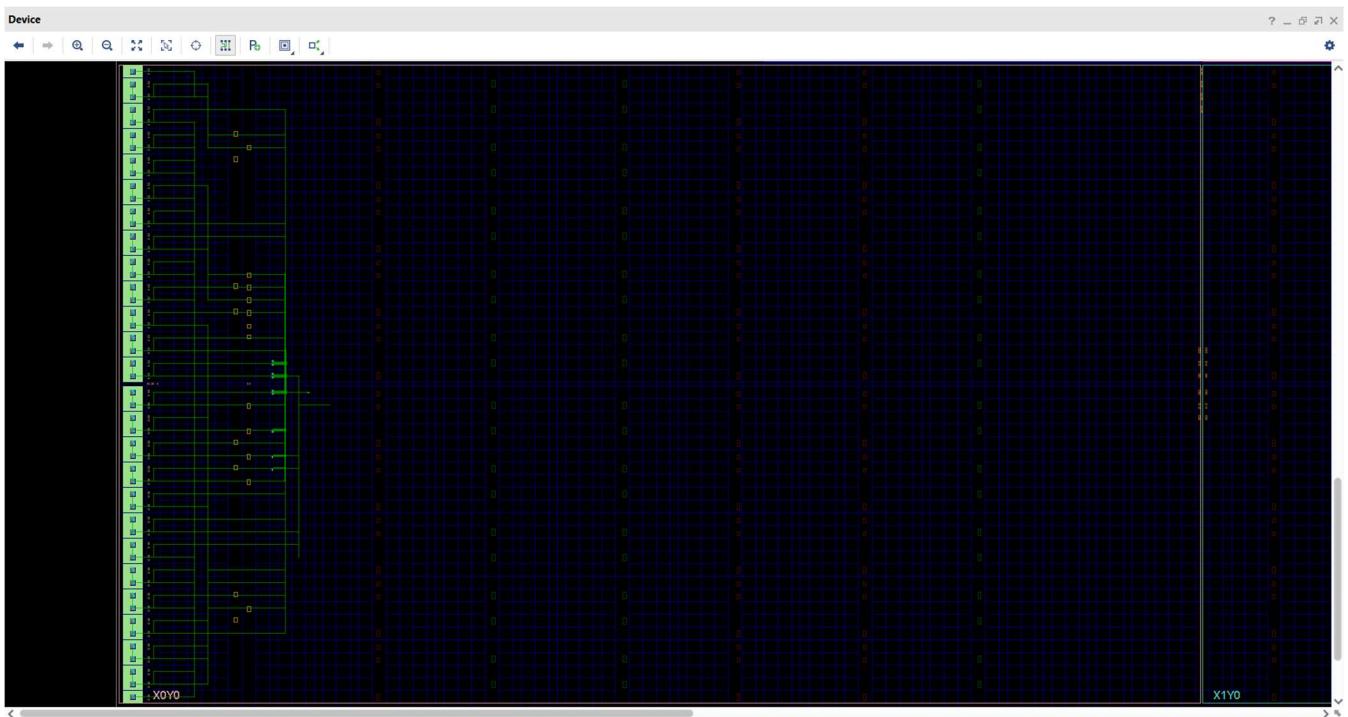
## Implementation – Device

Una volta sintetizzato il circuito posso procedere con l'implementazione del circuito su *Device* tramite *Run Implementation*. Nelle successive immagini [11](#), [12](#) e [13](#) si può notare come il circuito viene implementato su *Device* e soprattutto come tra le migliaia di *LUTs*<sup>4</sup> disponibili ne vengono utilizzate un numero piccolissimo.

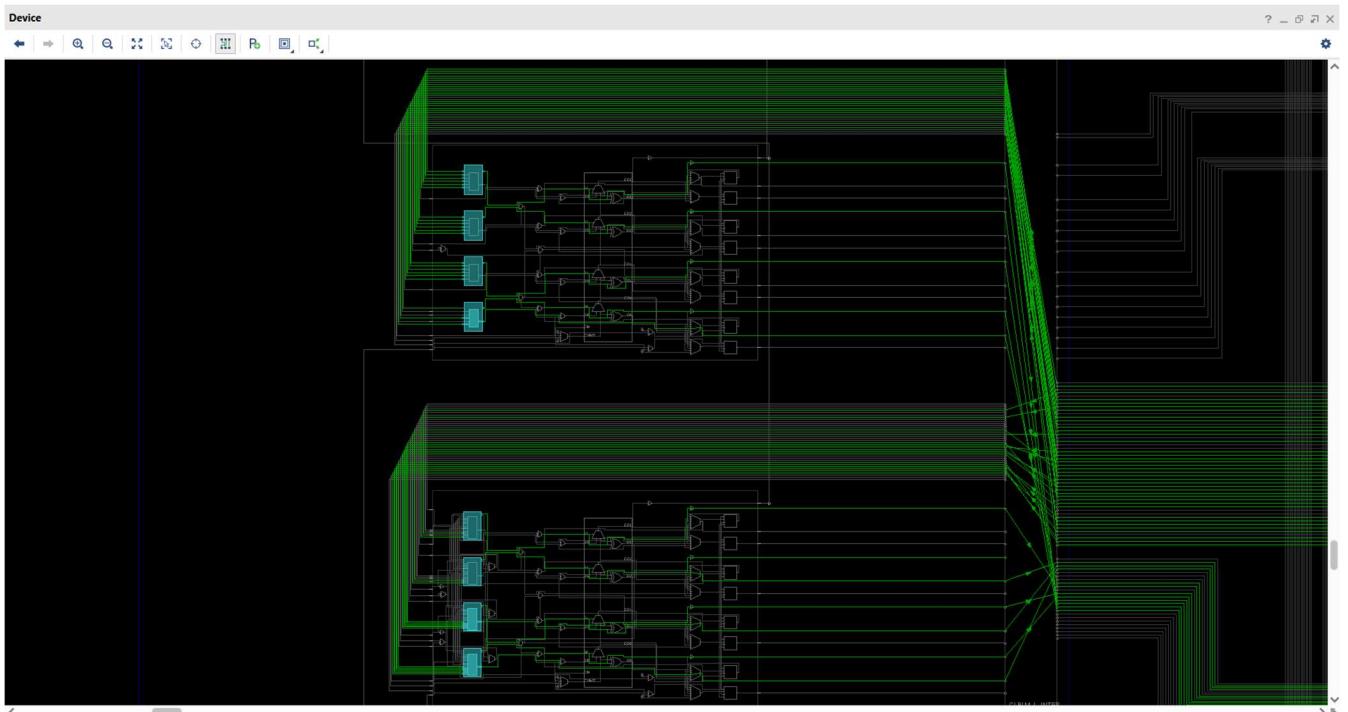


*Figura 11 Implementation - Device (1)*

<sup>4</sup> LUT è l'acronimo di Lookup Table, cioè una struttura che permette di associare ad ogni ammissibile combinazione di dati in ingresso una corrispondente configurazione di dati in uscita. Il termine inglese utilizzato per descriverle, *lookup table*, sottintende l'operazione di consultazione che permette di associare i dati in uscita ad una determinata combinazione dei dati in ingresso.



*Figura 12 Implementation - Device (2)*



*Figura 13 Implementation - Device (3)*

Analizzo, ora, il fattore di utilizzazione a fronte del processo di *Synthesis* e di *Implementation* del circuito in questione. Nella figura 15 posso notare come il processo di sintesi ha utilizzato 36 *LUTs* e 50 risorse *IO* tale che il fattore di utilizzazione risulta essere rispettivamente pari a 0.07% e 25% a fronte di 53200 e 200 risorse disponibili.

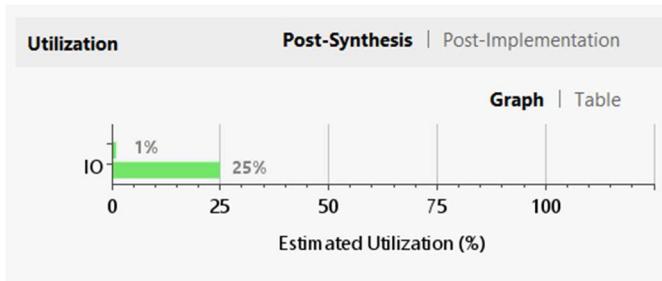


Figura 14 Post - Synthesis Utilization Graph

Utilization			
Post-Synthesis   Post-Implementation			
Graph   Table			
Resource	Estimation	Available	Utilization %
LUT	36	53200	0.07
IO	50	200	25.00

Figura 15 Post - Synthesis Utilization Table

Dopo aver effettuato, invece, l'implementazione posso notare che il fattore di utilizzazione delle *LUTs* risulta essere inferiore al fattore di utilizzazione della sintesi. Tanto è vero che esso è pari al 0.06%. Posso affermare che il Device utilizzato, il quale mette a disposizione ben 53200 *LUTs*, effettivamente utilizza in questo caso pochissime *LUTs*, cioè solo 34 *LUTs*.

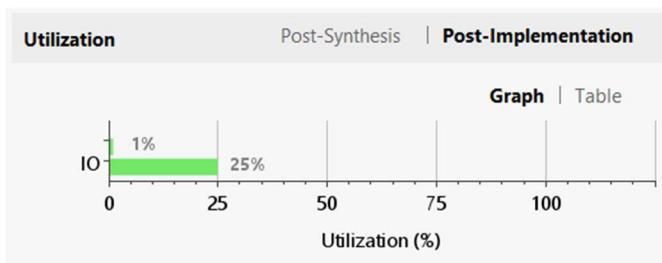


Figura 16 Post - Implementation Utilization Graph

Utilization			
Post-Synthesis   Post-Implementation			
Graph   Table			
Resource	Utilization	Available	Utilization %
LUT	34	53200	0.06
IO	50	200	25.00

Figura 15 Post - Implementation Utilization Table

Di seguito riporto i *Report* generati automaticamente da Vivado riguardo l'utilizzazione *Post - Implementation* del circuito in questione:

Site Type	Used	Fixed	Available	Util%
Slice LUTs	34	0	53200	0.06
LUT as Logic	34	0	53200	0.06
LUT as Memory	0	0	17400	0.00
Slice Registers	0	0	106400	0.00
Register as Flip Flop	0	0	106400	0.00
Register as Latch	0	0	106400	0.00
F7 Muxes	0	0	26600	0.00
F8 Muxes	0	0	13300	0.00

5

Figura 17 Post - Implementation Report Utilization (1)

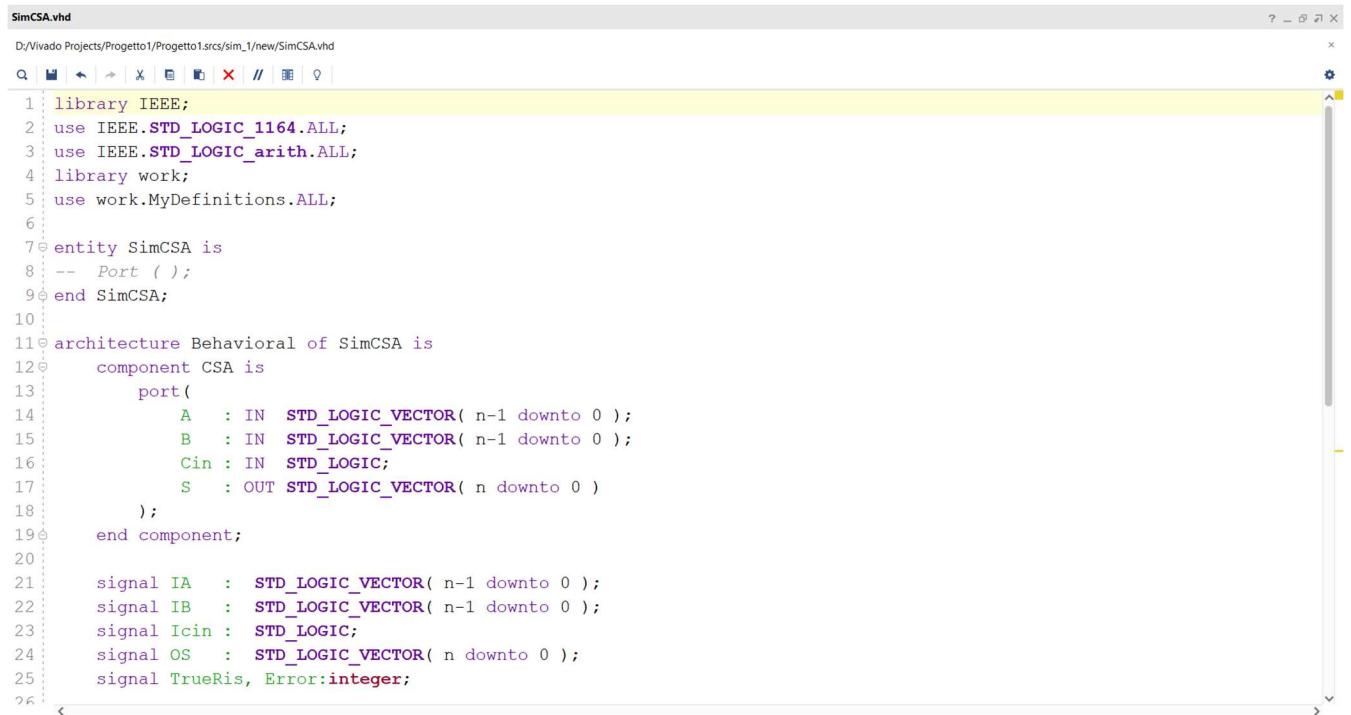
Site Type	Used	Fixed	Available	Util%
Slice	10	0	13300	0.08
SLICEL	3	0		
SLICEM	7	0		
LUT as Logic	34	0	53200	0.06
using O5 output only	0			
using O6 output only	21			
using O5 and O6	13			
LUT as Memory	0	0	17400	0.00
LUT as Distributed RAM	0	0		
LUT as Shift Register	0	0		
Slice Registers	0	0	106400	0.00
Register driven from within the Slice	0			
Register driven from outside the Slice	0			
Unique Control Sets	0		13300	0.00

Figura 18 Post - Implementation Report Utilization (2)

<sup>5</sup> Le Slice LUTs sono le LUTs utilizzabili

# SimCSA

Per quanto riguarda lo script di simulazione *SimCSA*, inizialmente inizializzo il *circuito CSA* esplicitando gli ingressi *A*, *B* e *Cin* e l'uscita *S*, dove *A* e *B* sono due vettori ognuno di *n* indici ed *S* è un vettore di *n+1* indici poiché esso conterrà anche il riporto in uscita del circuito. Successivamente, considero ben 6 *signal* dove i primi quattro (*IA*, *IB*, *ICin*, *OS*) sono dei vettori con le rispettive grandezze già esplicitate nel *component CSA* (tranne ovviamente *ICin*) e gli ultimi 2 *signal* (*TrueRis*, *Error*), inizializzati come *integer*, rappresenteranno rispettivamente il *risultato corretto* dell'operazione algebrica eseguita e lo *scarto di errore* che potrebbe esserci tra il risultato calcolato tramite circuito CSA e *TrueRis*.



The screenshot shows a VHDL code editor window titled "SimCSA.vhd". The code is written in VHDL and defines a component "CSA" with various ports and signals. The code is as follows:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
library work;
use work.MyDefinitions.ALL;

entity SimCSA is
-- Port ();
end SimCSA;

architecture Behavioral of SimCSA is
component CSA is
port(
    A : IN STD_LOGIC_VECTOR( n-1 downto 0 );
    B : IN STD_LOGIC_VECTOR( n-1 downto 0 );
    Cin : IN STD_LOGIC;
    S : OUT STD_LOGIC_VECTOR( n downto 0 )
);
end component;

signal IA : STD_LOGIC_VECTOR( n-1 downto 0 );
signal IB : STD_LOGIC_VECTOR( n-1 downto 0 );
signal Icin : STD_LOGIC;
signal OS : STD_LOGIC_VECTOR( n downto 0 );
signal TrueRis, Error:integer;
```

Figura 19 SimCSA (1)

Pertanto, inizializzo il circuito CSA con rispettivamente i *signal* *IA*, *IB*, *ICin* e *OS* e assegno a *ICin* il valore *std\_logic* '0'. Successivamente nel *process* effettuo una *simulazione esaustiva*, cioè effettuo una simulazione su tutti i casi possibili per vedere come si comporta il circuito in qualsiasi situazione. Tale simulazione la implemento con un doppio *for* scorrendo ognuno dai valori<sup>6</sup>  $-2^{n-1}$  ai valori  $2^{n-1} - 1$  e assegnando rispettivamente a *IA* e *IB* il valore *va-esimo* e *vb-esimo* per *n* bit tramite l'operazione *conv\_std\_logic\_vector*. Ovviamente il valore corretto *TrueRis* sarà ottenuto facendo una semplice addizione tra gli indici *va* e *vb*. Lo scarto di errore, invece, potrà essere ottenuto semplicemente tramite un'operazione di sottrazione tra il valore corretto *TrueRis* e il valore *OS*, ottenuto tramite l'operazione logica svolta dal circuito CSA e convertito dall'operazione *conv\_std\_logic\_vector* su *n* bit.

In questo caso, essendo che stiamo trattando il circuito CSA con caratteristiche reali a 16 bit, il range di valori in complemento a due sarà:

$$\begin{aligned} & [-2^{n-1}, 2^{n-1} - 1] \\ & [-2^{16-1}, 2^{16-1} - 1] \\ & [-2^{15}, 2^{15} - 1] \\ & [-32768, 32767] \end{aligned}$$

```

SimCSA.vhd
D:\Vivado Projects\Progetto1\Progetto1.srcs\sim_1\new\SimCSA.vhd
? _ ⊞ ⊖ ×
Q F ↻ ↺ ↻ X D // ⊞ Q
26
27
28
29 begin
30
31     csa_circ: CSA port map( IA, IB, Icin, OS );
32     Icin <= '0';
33
34 process
35 begin
36     for va in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
37         IA <= conv_std_logic_vector( va, n );
38         for vb in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
39             IB <= conv_std_logic_vector( vb, n );
40             TrueRis <= va+vb;
41             wait for 10ns;
42         end loop;
43     end loop;
44 end process;
45
46 Error <= TrueRis - conv_integer( signed(OS) );
47
48
49 end Behavioral;
50

```

Figura 20 SimCSA (2)

<sup>6</sup> Un numero binario ad *n* bit può rappresentare, tramite la rappresentazione in complemento a due, un range di valori compreso tra  $-2^{n-1}$  e  $2^{n-1} - 1$

## Behavioral Simulation

Posso notare che la *Behavioral Simulation* effettua una simulazione esaustiva sul circuito CSA con caratteristiche reali a 16 bit. Inoltre, posso notare come l'output *OS* e il risultato corretto *TrueRis* sono i medesimi tale che lo scarto di errore *Error* è pari a zero.

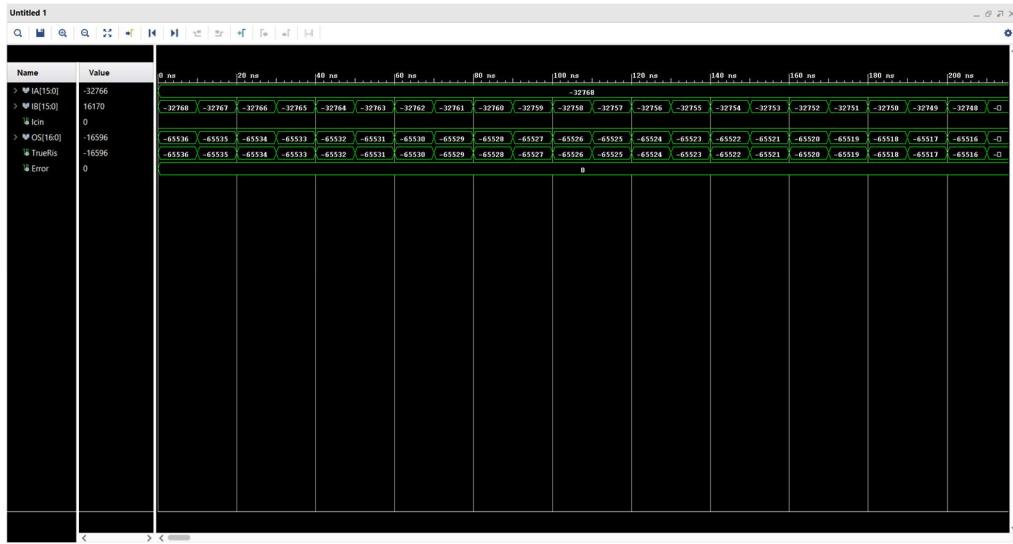


Figura 21 Behavioral Simulation da 0 ns a 180 ns con caratteristiche reali a 16 bit

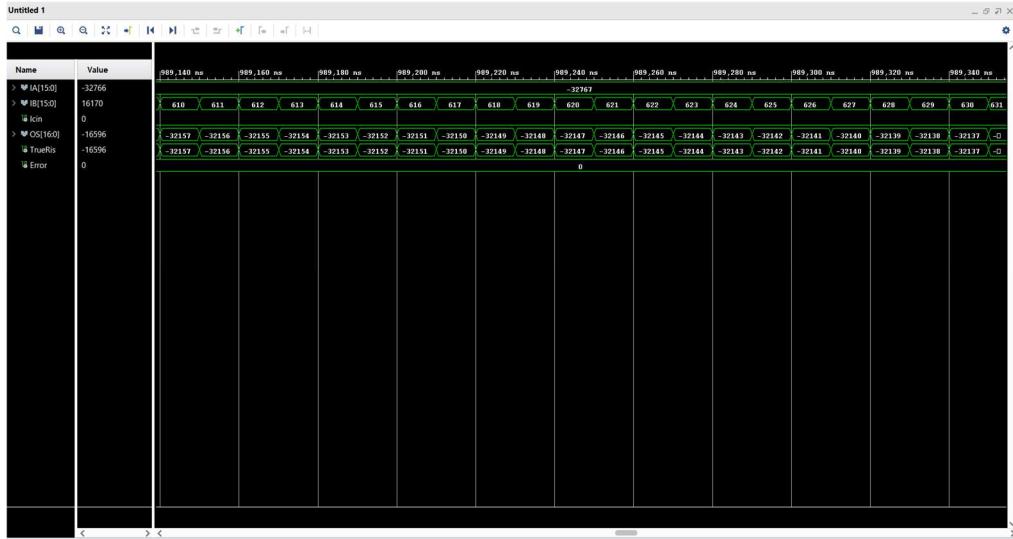


Figura 22 Behavioral Simulation da 989,140 ns a 989,340 ns con caratteristiche reali a 16 bit

## Post-Synthesis Timing Simulation

Avendo effettuato la Synthesis del circuito si può analizzare la Post-Synthesis Timing Simulation la quale sarà diversa dalla Behavioral Simulation. Infatti, posso notare che inizialmente sarà presente una *non specificazione* 'X' la quale sta a dire che inizialmente il software di simulazione sta ancora elaborando gli ingressi. Tanto è vero che per caratteristiche reali a 16 bit si avrà un ritardo pari a 3.692 ns tale che si avrà una discrepanza di errore pari a 1 in maniera *periodica* dovuto a tale ritardo.

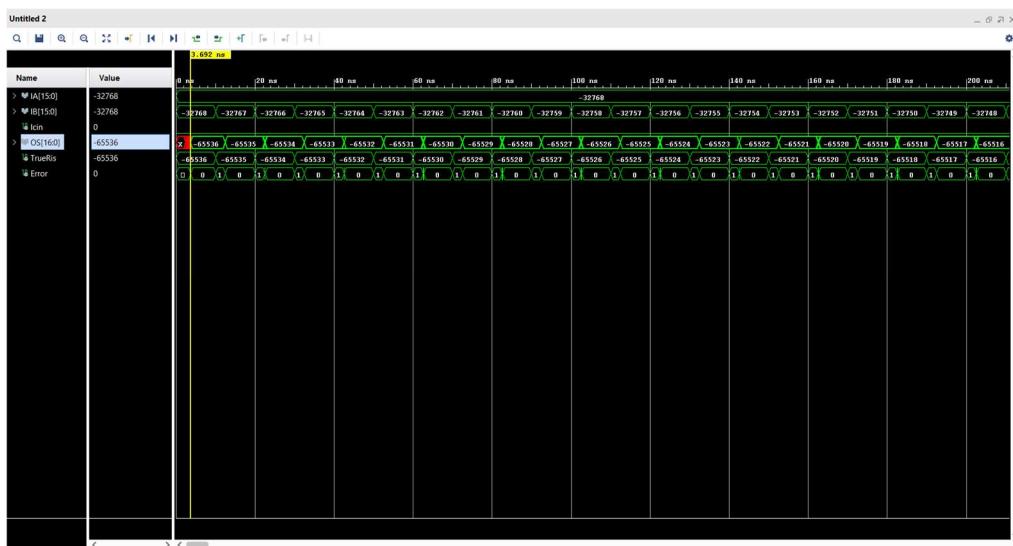


Figura 23 Post – Synthesis Timing Simulation da 0 ns a 180 ns con caratteristiche reali a 16 bit

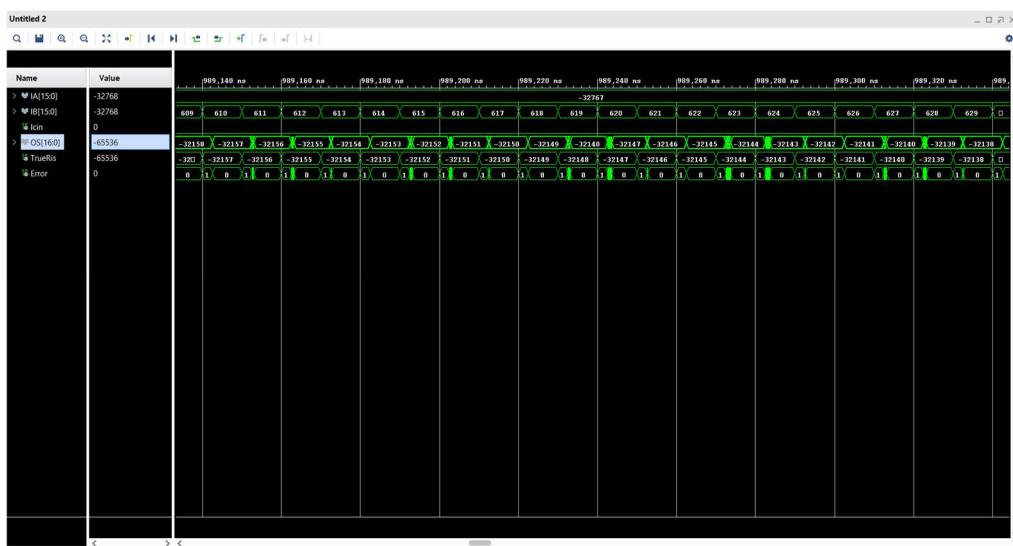


Figura 24 Post – Synthesis Timing Simulation da 989,140 ns a 989,320 ns con caratteristiche reali a 16 bit

## Post-Implementation Timing Simulation

Avendo effettuato l'implementazione del circuito CSA su *Device*, si può analizzare di conseguenza la Post-Implementation Timing Simulation. Infatti, posso notare un ritardo iniziale, dovuto sempre alla *non specificazione 'X'*, maggiore rispetto al ritardo ottenuto nella Post-Synthesis Timing Simulation. Questo ulteriore ritardo è dovuto all'implementazione del circuito su *Device*, cioè dovuto sia al ritardo dei componenti sia ai collegamenti implementati. Il comportamento ottenuto corrisponde al comportamento del Carry Select Adder se implementato su chip reale. Possiamo notare che il ritardo ottenuto nella Post-Implementation Timing Simulation con caratteristiche reali a 16 bit è pari a 6.317 ns tale che anche in questo caso sarà presente uno scarto di errore pari a 1 in maniera *periodica* dovuto al fatto che l'output *OS* del circuito non sarà *allineato* con il risultato corretto *TrueRis*. Quest'ultimo, ovviamente, non sarà affatto di ritardo essendo calcolato tramite una semplice operazione algebrica.

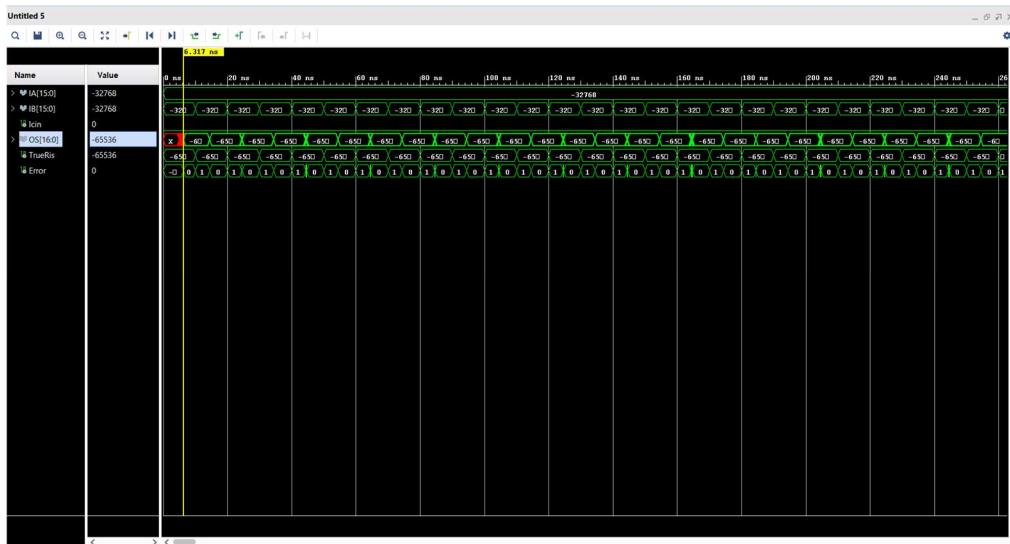


Figura 25 Post - Implementation Timing Simulation da 0 ns a 240 ns con caratteristiche reali a 16 bit

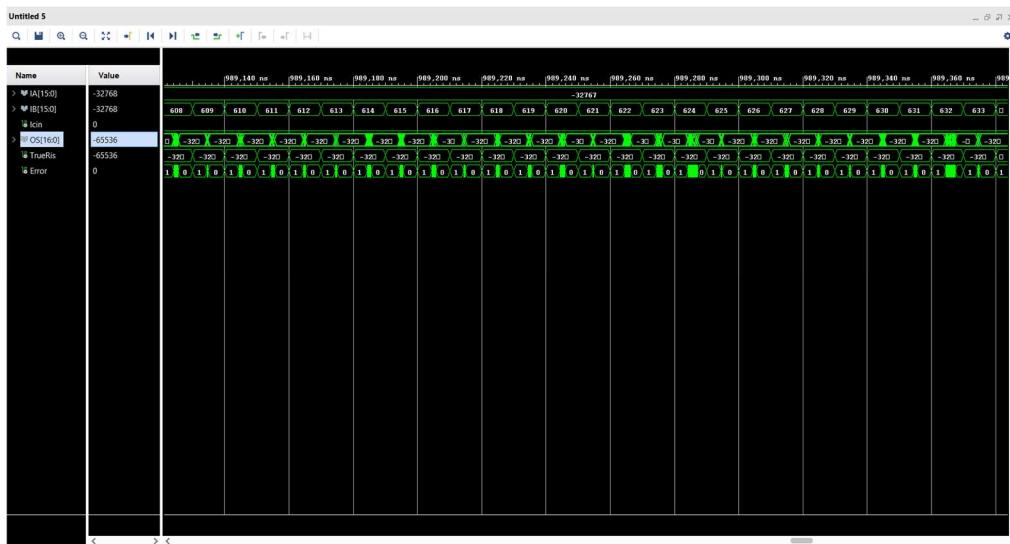


Figura 26 Post - Implementation Timing Simulation da 989,140 ns a 989,360 ns con caratteristiche reali a 16 bit

# Power On-Chip

Infine, posso analizzare il consumo di potenza da parte del circuito. La potenza totale su chip è la potenza consumata internamente all'interno dell'*FPGA*, pari alla somma della potenza statica del dispositivo e della potenza dinamica. Posso notare che l'8% della potenza totale è di tipo *static power*, cioè quella relativa allo stato stazionario dei segnali. La potenza statica del device è la potenza della dispersione del transistor su tutte le linee di tensione collegate e sui circuiti necessari per il normale funzionamento dell'*FPGA*, dopo la configurazione. Il 92% di consumo di potenza, invece, è di tipo *dynamic power*, cioè quella relativa alla dissipazione di potenza causata dalle commutazioni dei segnali. Questa potenza è instantanea e varia ad ogni ciclo di clock. Essa dipende dai livelli di tensione, dalla logica e dalle risorse di routing utilizzate. Il consumo di *dynamic power* è pari a  $P = CV^2F$  dove  $P$  è la *dynamic power* in Watt,  $V$  è la tensione di funzionamento in Volt,  $F$  è la frequenza in kHz ed, infine,  $C$  è la capacità di carico in  $\mu\text{F}$ . Una frequenza operativa più elevata comporta un consumo energetico dinamico più elevato. Solitamente, tutti i fornitori di *FPGA* forniscono stimatori per stimare la potenza dinamica in base all'utilizzo delle risorse e alla frequenza di commutazione. È importante fare una stima della potenza prima di selezionare un dispositivo per il sistema. Dal punto di vista del sistema, anche la gestione termica diventa una sfida quando il consumo energetico medio è elevato. In uno scenario del genere, se non si tiene conto della gestione termica, si potrebbe causare una fuga termica.

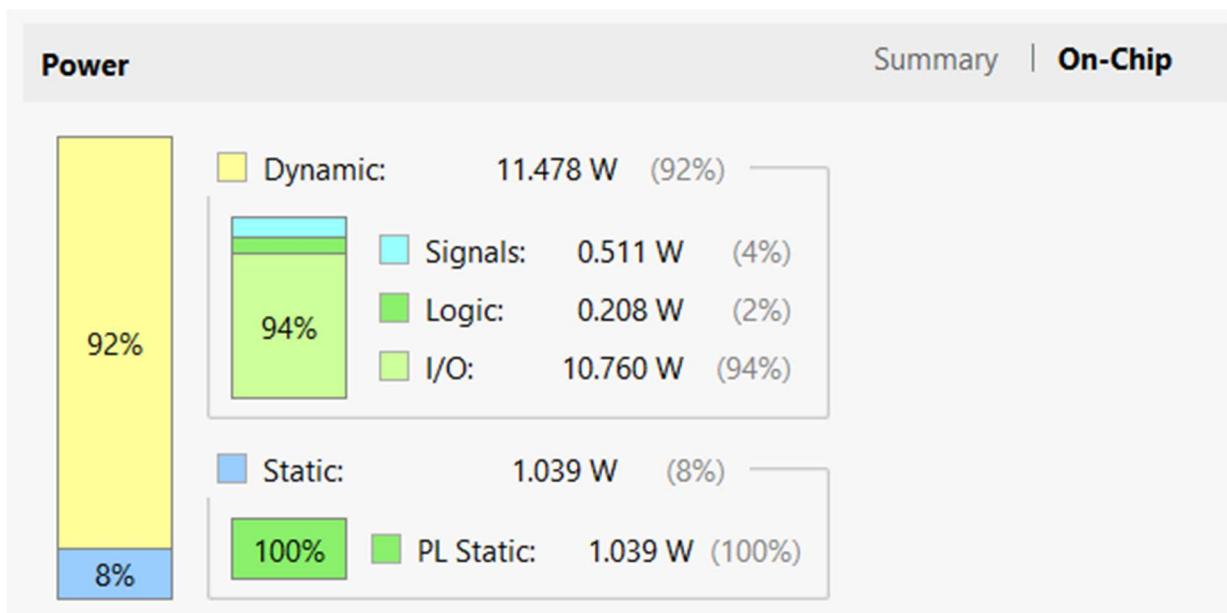


Figura 27 Power On-Chip