

# RELAZIONE DI PROGETTO: CARRY-SELECT ADDER A 16 BIT

Considerato che un **Carry-Select Adder a 16 bit** è composto da:

- 1 x **Ripple-Carry Adder a 4 bit**
- 3 x **Carry-Select Adder a 4 bit**

per il nostro progetto, si è scelto di realizzare il **Carry-Select Adder a 16 bit**, implementando inizialmente i blocchi che lo costituiscono.

La strategia è stata quella di partire dallo sviluppo dei blocchi più semplici, per arrivare progressivamente a quelli più complessi.

È stato approfondito di volta in volta lo studio di ciascun componente.

Si è iniziato con i **Ripple-Carry Adder a 4 bit**, ciascuno composto da:

- 4 x **Full-Adder**

dove un **Full-Adder** a 2 bit a sua volta è costituito da:

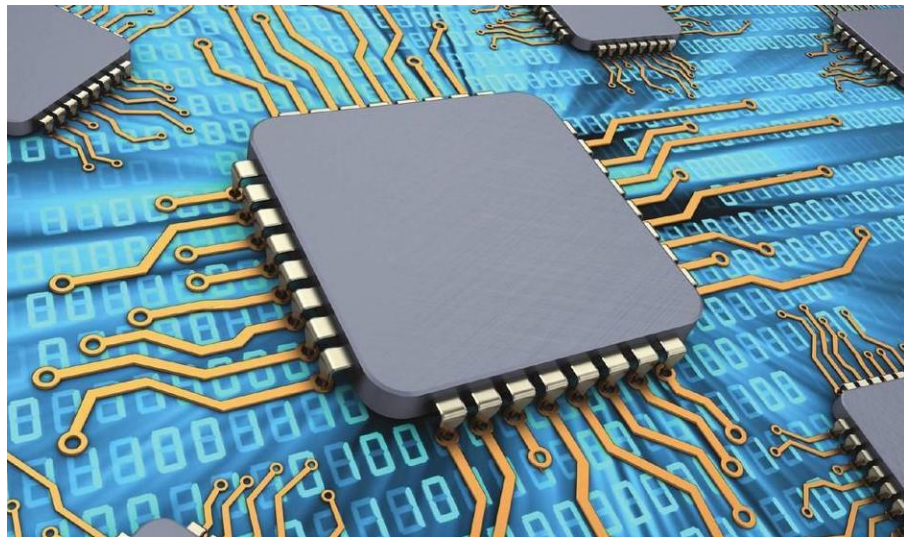
- 3 x porta AND
- 1 x porta OR
- 2 x porta XOR

Successivamente, sono stati descritti i **Carry-Select Adder a 4 bit**, ognuno formato da:

- 2 x **Ripple-Carry Adder a 4 bit**
- 5 x **Multiplexer 2\_1**

Pertanto, è stato definito il **Multiplexer 2\_1** con:

- 2 x porta AND
- 1 x porta OR
- 1 x porta NOT



# Definizioni

## RETI LOGICHE

Una **porta logica** è un circuito usato per realizzare in hardware una funzione logica elementare.

Un **circuito combinatorio** (o anche **rete combinatoria**) è un circuito il cui funzionamento riguarda solo la relazione ingresso-uscita. Tale relazione è descritta da una funzione logica.

Il **FAN-IN** indica qual è il massimo numero di ingressi che una funzione logica può processare senza commettere errori. In teoria una porta logica non ha limiti di input, ma in realtà dal punto di vista elettronico vi è una limitazione fisica: materialmente non è possibile avere più ingressi del parametro **FAN-IN**.

## LINGUAGGIO VHDL

Il linguaggio VHDL (*Very High Speed Integrated Circuits **H**ardware **D**escription **L**anguage*) è un metodo di descrizione delle funzioni logiche: è un linguaggio di descrizione (non di programmazione).

Esistono altri linguaggi di descrizione di circuiti oltre al VHDL, ma si differenziano solo per la sintassi: la semantica è la stessa.

Variabile = ausilio alla scrittura del codice. (*concetto astratto*)

Segnale = gestisce fisicamente il collegamento tra una porta logica ed un'altra.

TIPO di input/output

Il tipo **STANDARD LOGIC** amplia la possibilità di rappresentazione e per poterlo utilizzare bisogna importare ad inizio pagina la libreria in cui è presente:

```
library IEEE;    --rende visibile la libreria IEEE
```

```
use IEEE.STD_LOGIC_1164.ALL; --rende visibili i contenuti del package STD_LOGIC_1164
```

Per questo motivo quelli che sono stati chiamati **bit** per comodità, in realtà vengono definiti come **STD\_LOGIC** nel codice scritto.

## ENTITY

Nella **ENTITY** vi è la descrizione di ingressi e uscita.

## ARCHITECTURE

Nella **ARCHITECTURE** viene descritta la funzione da implementare: si utilizza il codice VHDL per indicare come sono rappresentate graficamente le varie componenti.

La keyword **OF** rappresenta il collegamento tra l'**ARCHITECTURE** e l'**ENTITY** sopra descritta.

### *COMPONENT*

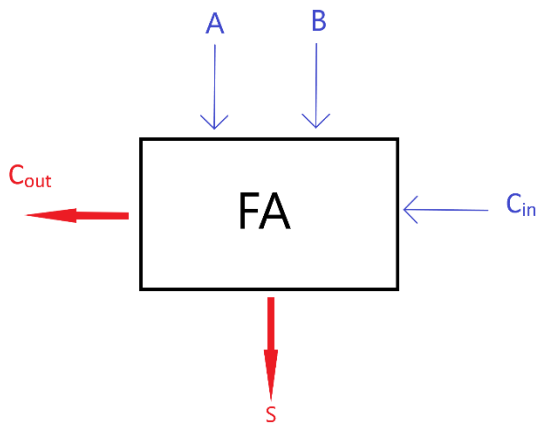
Il **VHDL** permette una modellazione gerarchica, ovvero è possibile assemblare un modulo attraverso sotto-moduli; ciò avviene con l'utilizzo della parola chiave **COMPONENT** che serve per richiamare i codici già scritti in altri file di testo.

### *PROCESS*

Il **PROCESS** aiuta a gestire la sequenzialità delle istruzioni, ovvero la loro dipendenza dal tempo.

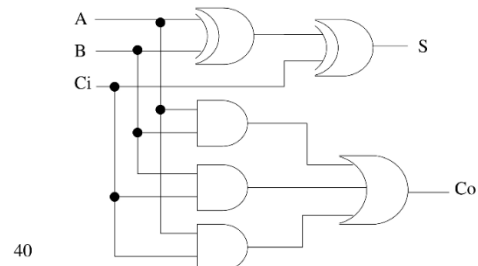
# Full-Adder

## Schema logico



## Circuito logico di un full-adder

$$C_o = B \cdot C_i + A \cdot C_i + A \cdot B \quad S = \bar{A} \cdot Y + A \cdot \bar{Y} = A \oplus B \oplus C_i$$



Il **Full-Adder** è un circuito logico caratterizzato da tre ingressi e due uscite.

Viene detto anche *sommatore completo*, in quanto può sommare due bit (**A** e **B** nel nostro caso) più un bit di riporto (CARRY) in ingresso (**C<sub>in</sub>**), seguendo la relazione:

$$A + B + C_{in} = S + C_{out}$$

restituendo quindi un bit somma (**S**) ed eventualmente un bit di riporto in uscita (**C<sub>out</sub>**).

In molti computer e altri tipi di processori, gli **adders** vengono utilizzati nell'unità aritmetico-logica o **ALU**.

La somma binaria è l'operazione di base, le altre operazioni derivano da essa.

Pertanto, se si riesce a creare un ottimo sommatore in un circuito, si possono svolgere tutte le operazioni.

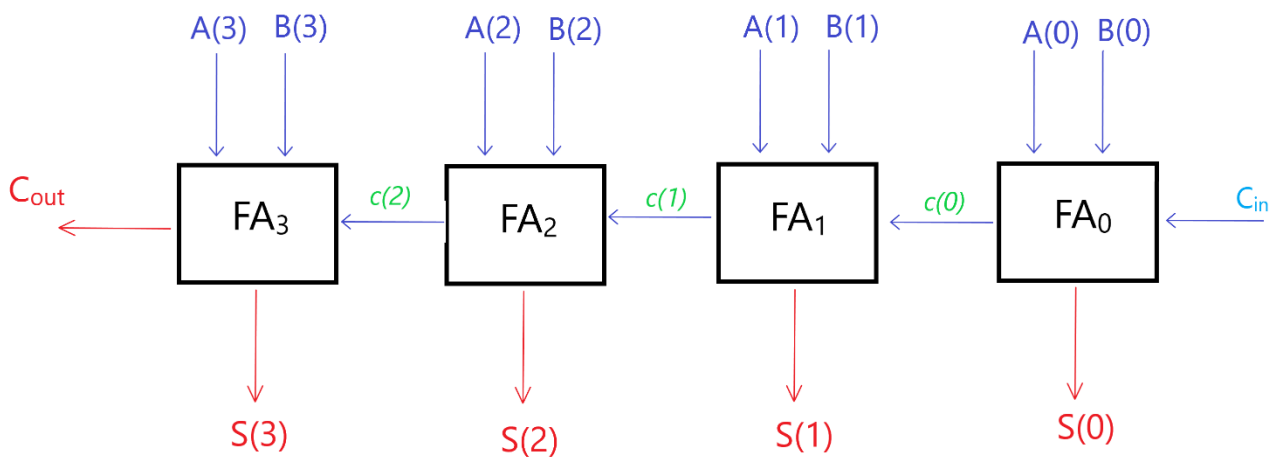
Nel progetto, è stato implementato il **Full-Adder** secondo lo schema della **slide 40** riportata in alto a destra.

```
FA.vhd
C:/Users/anton/Documents/RACCOLTE/DOCUMENTI/vvado-vhdl workspace/CarrySelectAdder16bit/CarrySelectAdder16bit.srscs/sources_1/new/FA.vhd

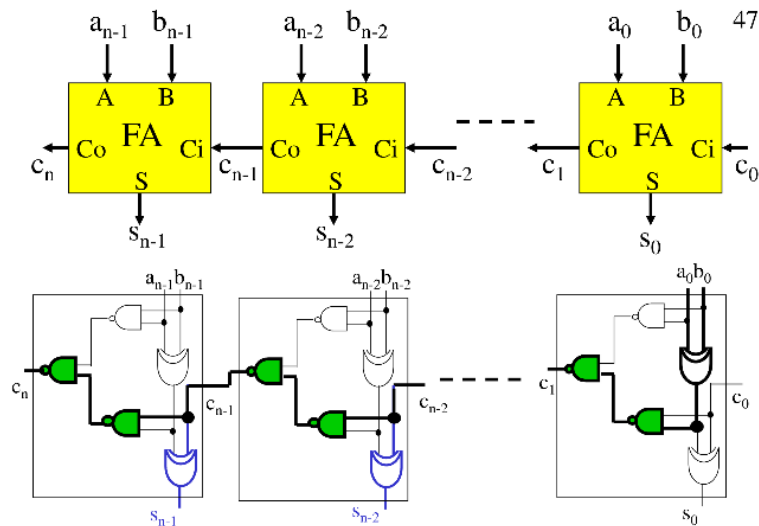
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity FA is
5     Port ( A : in STD_LOGIC; --ingresso A
6           B : in STD_LOGIC; --ingresso B
7           Cin : in STD_LOGIC; --CARRY IN (in ingresso)
8           S : out STD_LOGIC; --somma S tra A e B
9           Cout : out STD_LOGIC); --CARRY OUT (in uscita)
10 end FA;
11
12 architecture Behavioral of FA is
13
14 begin
15     S <= A XOR B XOR Cin ;
16     Cout <= (A AND B) OR (A AND Cin) OR (B AND Cin);
17 end Behavioral;
```

# Ripple-Carry Adder a 4 bit

## Schema logico



## Circuito logico di un Ripple-Carry a $n$ bit



Il metodo più diretto per realizzare un addizionatore ad  $n$  bit (nel nostro caso a 4 bit) è rappresentato dal **Ripple-Carry Adder** (RCA) o addizionatore a propagazione del riporto.

Questo circuito richiede  $n$  (nel nostro caso 4) **Full-Adder** in cascata, ovvero con il riporto uscente dell' $i$ -esimo **Full-Adder** collegato al riporto entrante dell' $(i+1)$ -esimo **Full-Adder**, come mostrato nello schema alla [slide 47](#) e sopra riportato.

L'addizionatore RCA non fa nient'altro che calcolare la somma così come verrebbe calcolata secondo il metodo *carta e penna*.

La sua architettura è semplice, ma anche lenta. Il tempo di calcolo nel caso peggiore, infatti, dipende linearmente dal numero di stadi, in quanto bisogna attendere che il riporto si propaghi dalla prima all'ultima cella per avere il risultato corretto.

Le considerazioni sul funzionamento di questo componente hanno suggerito di definirlo partendo dal **Full-Adder**, già implementato.

In particolare, sono stati utilizzati:

- Due ingressi (*operandi*) a 4 bit (**A** e **B**), dei quali ogni bit sarà ingresso del rispettivo **Full-Adder** in cascata.
- Un bit di riporto (CARRY) in ingresso (**C<sub>in</sub>**).
- Un'uscita somma (**S**), sempre a 4 bit, della quale ogni bit sarà uscita del rispettivo **Full-Adder** in cascata.
- Un bit di riporto in uscita (**C<sub>out</sub>**).

Oltre all'implementazione generale, è stata utilizzata una *signal* che viene anche riportata in [figura](#):

- Un segnale **c** creato come **STD\_LOGIC\_VECTOR** a 3 bit.

Questo segnale viene creato a 3 bit e non a 4, poiché l'ipotetico  $c(3)$  sarebbe dovuto essere assegnato al **C<sub>out</sub>** del **Ripple-Carry Adder** ma, per come è stato implementato, si è risparmiato l'utilizzo del quarto bit, assegnando l'uscita dell'ultimo **Full-Adder** direttamente al **C<sub>out</sub>**: questo lo si fa per evitare il fenomeno del **TRIMMING**.

```

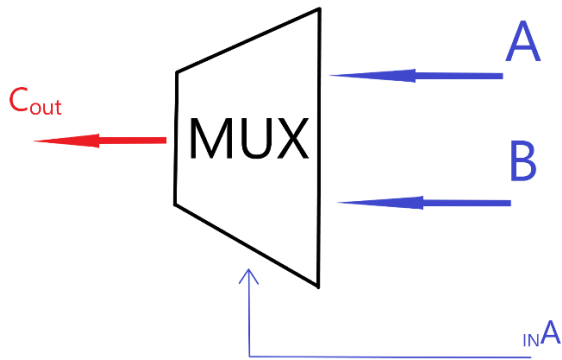
RCA4.vhd
C:/Users/antonio/Documents/RACCOLTE/DOCUMENTI/vivado-vhdl workspace/CarrySelectAdder16bit/CarrySelectAdder16bit/srcs/sources_1/new/RCA4.vhd

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- un Ripple-Carry Adder è costituito da 4 Full-Adder in cascata.
5
6 entity RCA4 is
7     Port ( A : in STD_LOGIC_VECTOR (3 downto 0); --ingresso A del Ripple-Carry a 4 bit
8           B : in STD_LOGIC_VECTOR (3 downto 0); --ingresso B del Ripple-Carry a 4 bit
9           Cin : in STD_LOGIC; --CARRY IN (in ingresso)
10          S : out STD_LOGIC_VECTOR (3 downto 0); --somma S tra A e B
11          Cout : out STD_LOGIC; --CARRY OUT (in uscita)
12 end RCA4;
13
14 architecture Behavioral of RCA4 is
15
16     signal c : STD_LOGIC_VECTOR(2 downto 0); --CARRY dei Full-Adder
17
18     component FA
19         Port ( A : in STD_LOGIC;
20               B : in STD_LOGIC;
21               Cin : in STD_LOGIC;
22               S : out STD_LOGIC;
23               Cout : out STD_LOGIC);
24     end component;
25
26     begin
27         FA0 : FA
28             Port map( A(0), B(0), Cin, S(0), c(0));
29         FA1 : FA
30             Port map( A(1), B(1), c(0), S(1), c(1));
31         FA2 : FA
32             Port map( A(2), B(2), c(1), S(2), c(2));
33         FA3 : FA
34             Port map( A(3), B(3), c(2), S(3), Cout); --Cout sarebbe c(3)
35
36     end Behavioral;

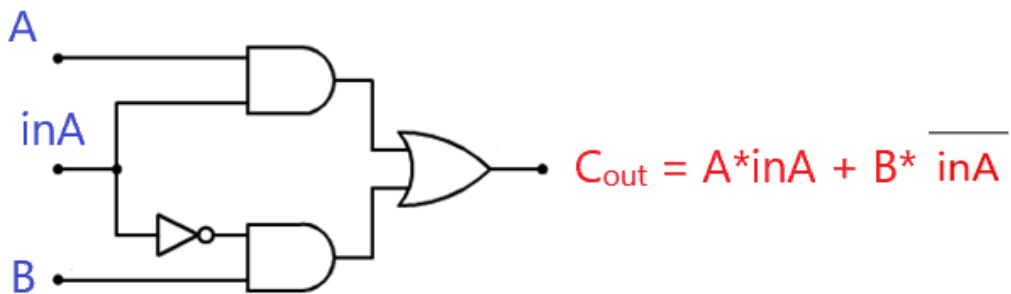
```

# Multiplexer

## Schema logico



## Circuito logico di un Multiplexer 2 a 1

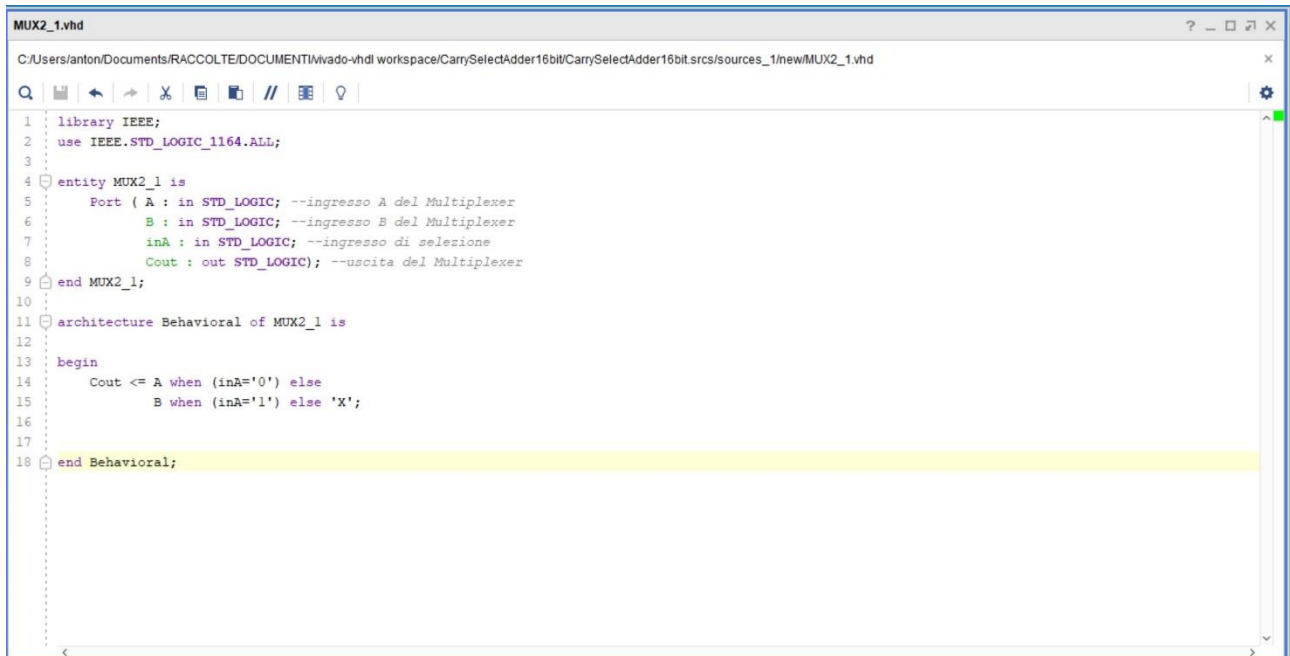


Un **Multiplexer** (comunemente detto **MUX**) è un circuito logico che consente di selezionare (**selettore**) 1 tra  $2^n$  ingressi in base allo stato di  $n$  segnali di controllo.

Sono soprattutto usati per aumentare la quantità di dati che possono essere trasmessi attraverso una rete in determinate quote di tempo e banda, pertanto il termine viene usato, contestualmente, sia in **elettronica** che in telecomunicazioni.



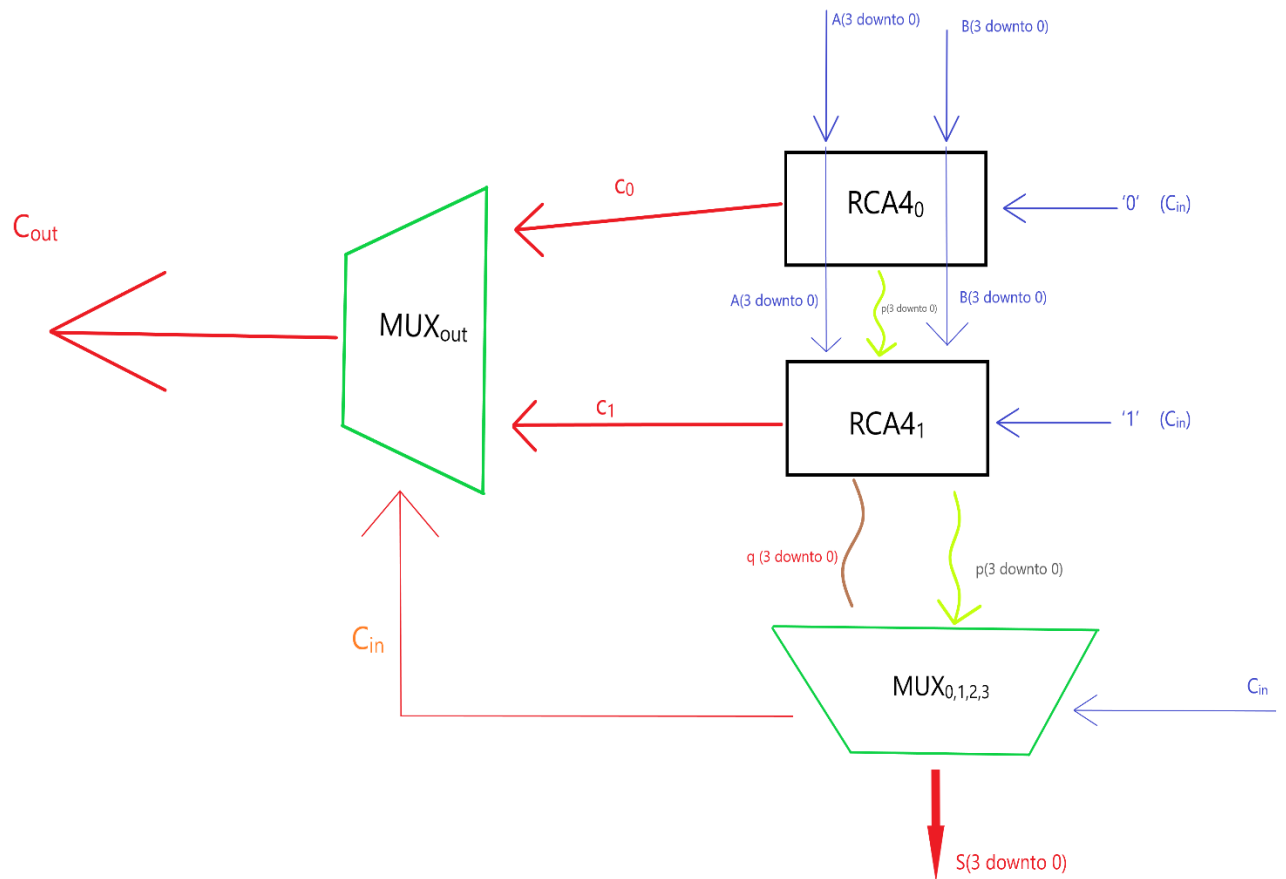
Come si può osservare nella **figura rappresentante il circuito logico**, il **MUX** riceve in ingresso due bit (chiamati **A** e **B** nel nostro codice VHDL) e un bit di selezione (**inA**) che determina quale dei due ingressi far passare in uscita al circuito (**Cout**).



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity MUX2_1 is
5      Port ( A : in STD_LOGIC; --ingresso A del Multiplexer
6            B : in STD_LOGIC; --ingresso B del Multiplexer
7            inA : in STD_LOGIC; --ingresso di selezione
8            Cout : out STD_LOGIC); --uscita del Multiplexer
9  end MUX2_1;
10
11  architecture Behavioral of MUX2_1 is
12
13  begin
14      Cout <= A when (inA='0') else
15              B when (inA='1') else 'X';
16
17
18  end Behavioral;
```

# Carry-Select Adder a 4 bit

## Schema logico



In elettronica, un **Carry-Select Adder** è un circuito che rappresenta un modo particolare per implementare un generico **adder**.

La sua struttura è piuttosto semplice, ma comunque abbastanza veloce nel calcolo.

Dalla definizione di questo componente, dopo aver valutato lo schema logico **sopra riportato**, si è deciso di strutturarne partendo dal **Ripple-Carry Adder** già implementato (2 unità per il **Carry-Select** a 4 bit), e utilizzando 5 **multiplexer**.

In particolare, l'implementazione generale scelta riguarda:

- Due ingressi (*operandi*) a 4 bit (**A** e **B**) dei due **Ripple-Carry** connessi in parallelo.
- Un bit di riporto in ingresso (**C<sub>in</sub>**).
- Un'uscita somma (**S**), sempre a 4 bit.
- Un bit di riporto in uscita (**C<sub>out</sub>**).

Oltre all'implementazione generale, sono state definite delle *signal* che vengono anche riportate in **figura**:

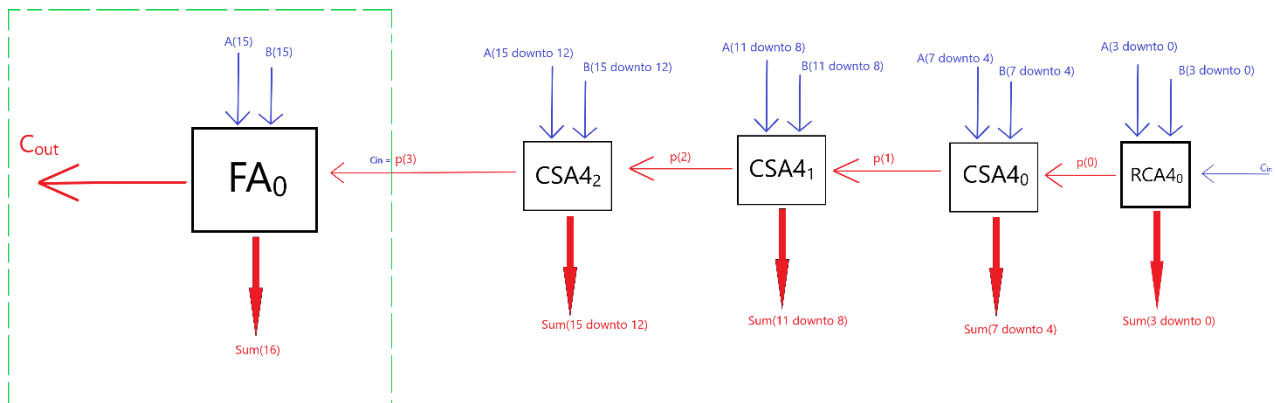
- Due segnali **p** e **q** creati come **STD\_LOGIC\_VECTOR** a 4 bit, dei quali ogni bit sarà uscita del rispettivo Full-Adder nel Ripple-Carry (**p** per il Ripple-Carry che riceve  $C_{in}=0$  e **q** per il Ripple-Carry che riceve  $C_{in}=1$ ), che saranno scelti come bit finali di **S** tramite **multiplexer**.
- Due segnali **c0** e **c1** creati come **STD\_LOGIC**, usati come riporti per i rispettivi Ripple-Carry.

```
CSA4.vhd
C:/Users/anton/Documents/RACCOLTE/DOCUMENTI/mvado-vhdl workspace/CarrySelectAdder16bit/CarrySelectAdder16bit/srcs/sources_1/new/CSA4.vhd

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  --un Carry-Select Adder a 4 bit è costituito da 2 Ripple-Carry Adder a 4 bit e 5 Multiplexer.
5
6  entity CSA4 is
7      Port ( A : in STD_LOGIC_VECTOR (3 downto 0); --ingresso A del Carry-Select Adder a 4 bit
8            B : in STD_LOGIC_VECTOR (3 downto 0); --ingresso B del Carry-Select Adder a 4 bit
9            Cin : in STD_LOGIC; --CARRY IN (in ingresso)
10           S : out STD_LOGIC_VECTOR (3 downto 0); --somma S tra A e B
11           Cout : out STD_LOGIC; --CARRY OUT (in uscita)
12 end CSA4;
13
14 architecture Behavioral of CSA4 is
15
16     signal p, q : STD_LOGIC_VECTOR(3 downto 0); -- somme in uscita dei Full-Adder all'interno dei Ripple-Carry Adder
17     signal c0, c1 : STD_LOGIC; --CARRY OUT (in uscita) rispettivamente di RCA4_0 e RCA4_1
18
19     component MUX2_1
20         Port( A : in STD_LOGIC;
21              B : in STD_LOGIC;
22              inA : in STD_LOGIC;
23              Cout: out STD_LOGIC);
24     end component;
25
26     component RCA4
27         Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
28              B : in STD_LOGIC_VECTOR (3 downto 0);
29              Cin : in STD_LOGIC;
30              S : out STD_LOGIC_VECTOR (3 downto 0);
31              Cout : out STD_LOGIC);
32     end component;
33
34     begin
35
36     -- Ripple-Carry Adder
37     RCA4_0 : RCA4
38         Port map(A(3 downto 0), B(3 downto 0), '0', p(3 downto 0), c0);
39     RCA4_1 : RCA4
40         Port map(A(3 downto 0), B(3 downto 0), '1', q(3 downto 0), c1);
41
42     -- Multiplexer delle somme dei Ripple-Carry Adder
43     MUX0 : MUX2_1
44         Port map(p(0), q(0), Cin, S(0));
45     MUX1 : MUX2_1
46         Port map(p(1), q(1), Cin, S(1));
47     MUX2 : MUX2_1
48         Port map(p(2), q(2), Cin, S(2));
49     MUX3 : MUX2_1
50         Port map(p(3), q(3), Cin, S(3));
51
52     -- Multiplexer dei CARRY dei due Ripple-Carry Adder
53     MUX_OUT : MUX2_1
54         Port map(c0, c1, Cin, Cout);
55
56 end Behavioral;
```

# Carry-Select Adder a 16 bit

## Schema logico



Come nei casi precedenti, un **Carry-Select Adder** a 16 bit può essere visto come composizione di più circuiti più semplici, in questo caso tre blocchi di **Carry-Select Adder** a 4 bit e un **Ripple-Carry** a 4 bit.

L'implementazione scelta consiste di:

- Due ingressi (*operandi*) a 16 bit (**A** e **B**) del **Ripple-Carry** e dei **Carry-Select Adder** a 4 bit.
- Un bit di riporto in ingresso (**C<sub>in</sub>**), inserito come ingresso del **Ripple-Carry**.
- Un'uscita somma (**S**), questa volta a 17 bit per gestire il problema dell'*overflow* nella somma di bit espressi in *complemento a 2*.
- Un bit di riporto in uscita (**C<sub>out</sub>**).

Come nel caso del **Carry-Select Adder** a 4 bit, sono state usate due *signal* per gestire i componenti in serie:

- Segnale **p**, uno *STD\_LOGIC\_VECTOR* a 4 bit, usato per i riporti in-out dei componenti.
- Segnale **Sum**, uno *STD\_LOGIC\_VECTOR* che raccoglie le somme parziali di ogni blocco. Come si nota a **riga 59** del codice di CSA16 in VHDL, assegniamo Sum a S, e ciò è possibile poiché i due *STD\_LOGIC\_VECTOR* sono della stessa lunghezza.

Alla fine della rete, inoltre, è stato posto un **Full-Adder** per evitare l'errore di overflow, seguendo la definizione:

*Per somme algebriche di numeri binari rappresentati in complemento a 2 si verifica overflow quando:*

1. *la somma tra due numeri entrambi positivi restituisce un numero negativo;*
2. *la somma tra due numeri entrambi negativi restituisce un numero positivo.*

Il **Full-Adder** riceve in ingresso come:

- “A”,  $A(15)$ , ovvero l'MSB dell'operando A.
- “B”,  $B(15)$ , ovvero l'MSB dell'operando B.
- “Cin”, il *Cout* del CSA4<sub>2</sub>, ovvero il segnale di riporto dell'ultimo **Carry-Select a 4 bit**.

Invece, come uscita avremo come:

- “S”,  $Sum(16)$ , ovvero l'MSB della somma totale  $Sum(16 \text{ downto } 0)$ .
- “Cout”, il riporto finale della somma dei due operandi in *complemento a 2*.

```

CSA16.vhd
C:\Users\anton\Documents\RACCOLTE\DOCUMENTI\vhdl\workspace\Carr\SelectedAdder16bit\Carr\SelectedAdder16bit\srcs\srcs_new\CSA16.vhd

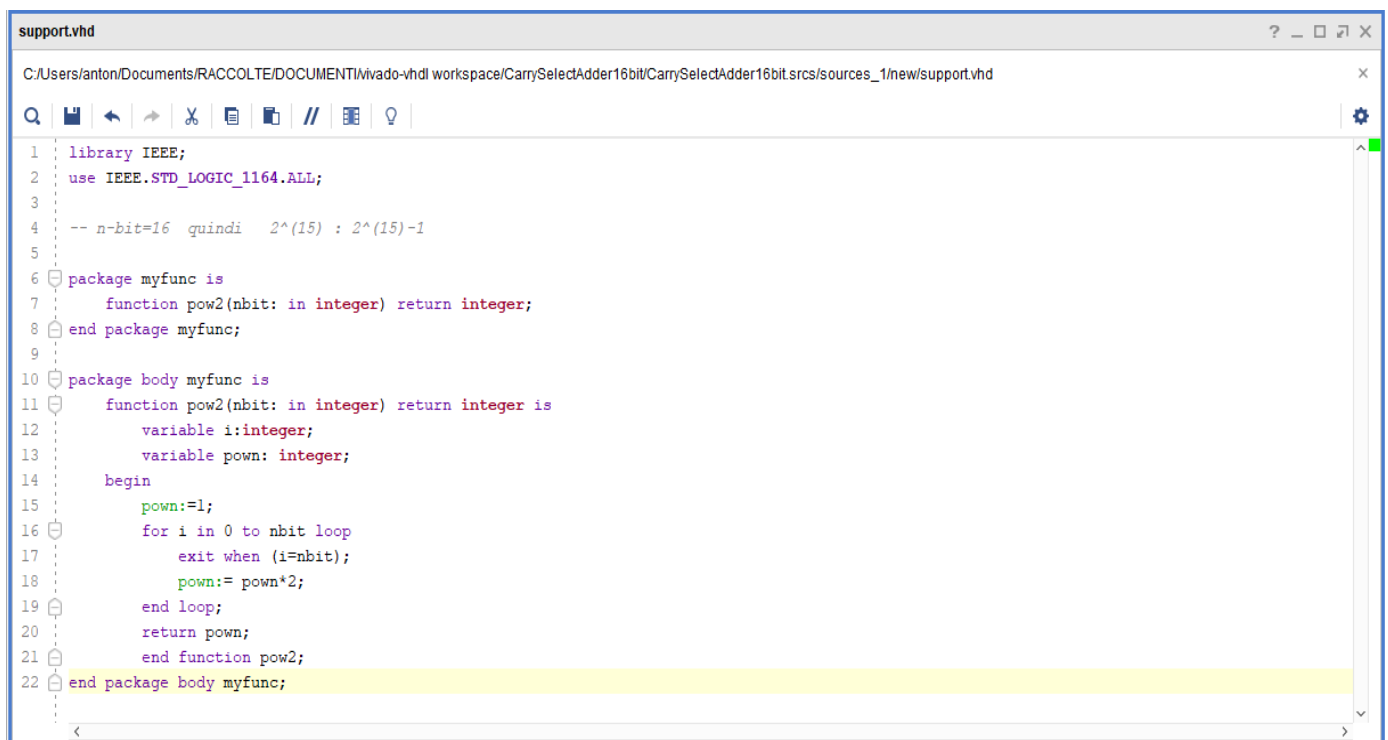
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- un Carry-Select Adder a 16 bit è costituito da 3 Carry-Select Adder a 4 bit e 4 Full-Adder (ovvero un Ripple-Carry Adder a 4 bit).
5
6 entity CSA16 is
7     Port ( A : in STD_LOGIC_VECTOR (15 downto 0); --ingresso A del Carry-Select Adder a 16 bit
8           B : in STD_LOGIC_VECTOR (15 downto 0); --ingresso B del Carry-Select Adder a 16 bit
9           Cin : in STD_LOGIC; --CARRY IN (in ingresso)
10          S : out STD_LOGIC_VECTOR (16 downto 0); --somma S tra A e B
11          Cout : out STD_LOGIC; --CARRY OUT (in uscita)
12 end CSA16;
13
14 architecture Behavioral of CSA16 is
15
16     signal p : STD_LOGIC_VECTOR (3 downto 0);
17     signal Sum : STD_LOGIC_VECTOR (16 downto 0); --segnale delle somme parziali
18
19     component FA
20         Port ( A : in STD_LOGIC;
21               B : in STD_LOGIC;
22               Cin : in STD_LOGIC;
23               S : out STD_LOGIC;
24               Cout : out STD_LOGIC);
25     end component;
26
27     component RCA4
28         Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
29               B : in STD_LOGIC_VECTOR (3 downto 0);
30               Cin : in STD_LOGIC;
31               S : out STD_LOGIC_VECTOR (3 downto 0);
32               Cout : out STD_LOGIC);
33     end component;
34
35     component CSA4
36         Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
37               B : in STD_LOGIC_VECTOR (3 downto 0);
38               Cin : in STD_LOGIC;
39               S : out STD_LOGIC_VECTOR (3 downto 0);
40               Cout : out STD_LOGIC);
41     end component;
42
43
44     begin
45         --Ripple-Carry Adder a 4 bit
46         RCA4_0 : RCA4
47             Port map (A(3 downto 0), B(3 downto 0), Cin, Sum(3 downto 0), p(0));
48         --Carry-Select Adder a 4 bit
49         CSA4_0 : CSA4
50             Port map (A(7 downto 4), B(7 downto 4), p(0), Sum(7 downto 4), p(1));
51         CSA4_1 : CSA4
52             Port map (A(11 downto 8), B(11 downto 8), p(1), Sum(11 downto 8), p(2));
53         CSA4_2 : CSA4
54             Port map (A(15 downto 12), B(15 downto 12), p(2), Sum(15 downto 12), p(3));
55
56         -- Full Adder per gestire il riporto
57         FA0 : FA
58             Port map (A(15), B(15), p(3), Sum(16), Cout);
59         S <= Sum;
60
61     end Behavioral;

```

# support

In VHDL non esiste una funzione che calcoli  $2^n$ , dunque è stata definita con il nome di *myfunc* dentro il file di testo *support*.

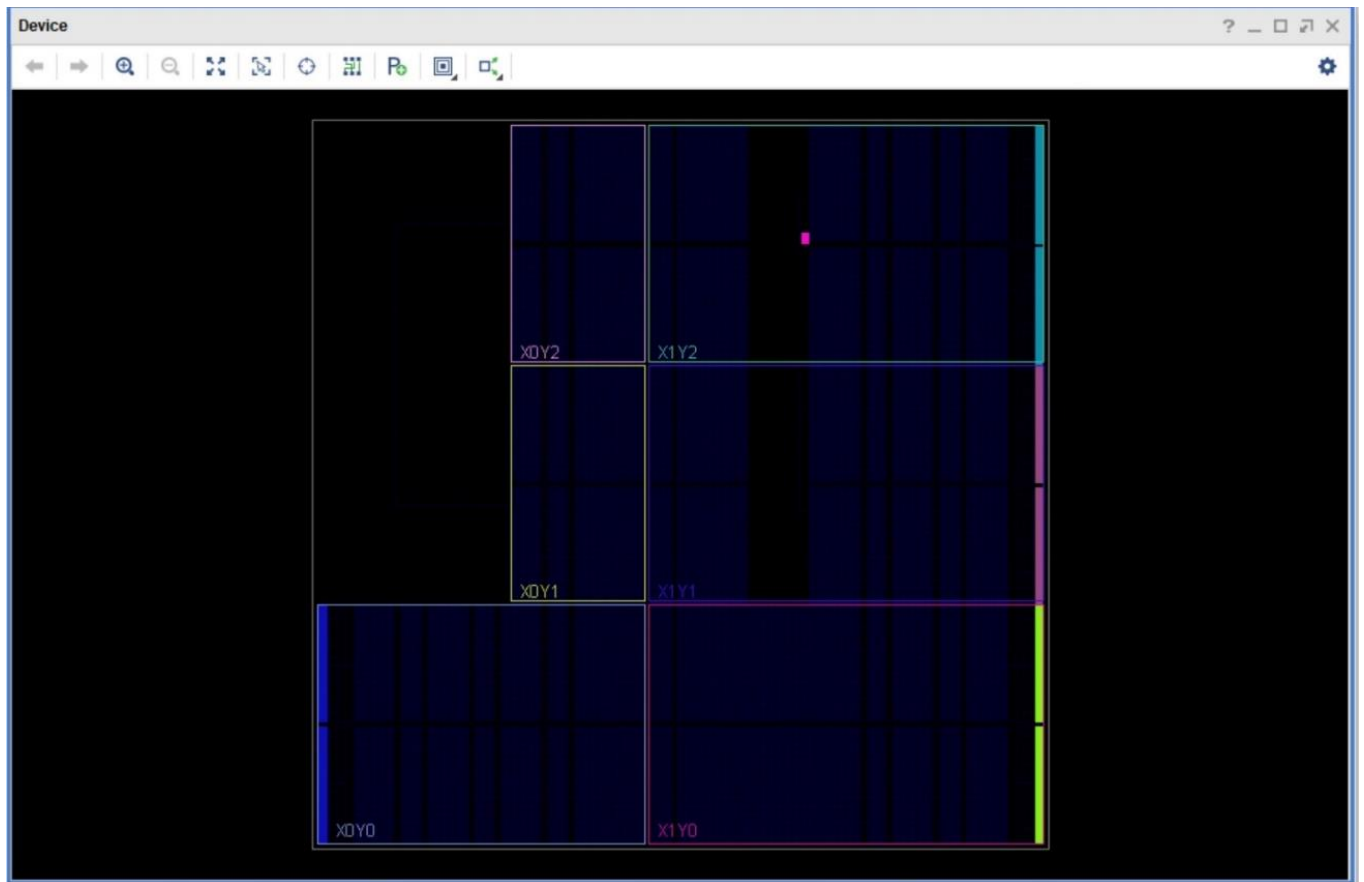
All'interno della *begin* si utilizza il **for loop**, il quale permette di eseguire per *n* volte un gruppo di statement sequenziali.



```
support.vhd
C:/Users/anton/Documents/RACCOLTE/DOCUMENTI/vivado-vhdl workspace/CarrySelectAdder16bit/CarrySelectAdder16bit.srscs/sources_1/new/support.vhd

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- n-bit=16 quindi 2^(15) : 2^(15)-1
5
6  package myfunc is
7      function pow2(nbit: in integer) return integer;
8  end package myfunc;
9
10 package body myfunc is
11     function pow2(nbit: in integer) return integer is
12         variable i:integer;
13         variable pown: integer;
14     begin
15         pown:=1;
16         for i in 0 to nbit loop
17             exit when (i=nbit);
18             pown:= pown*2;
19         end loop;
20         return pown;
21     end function pow2;
22 end package body myfunc;
```

# Device

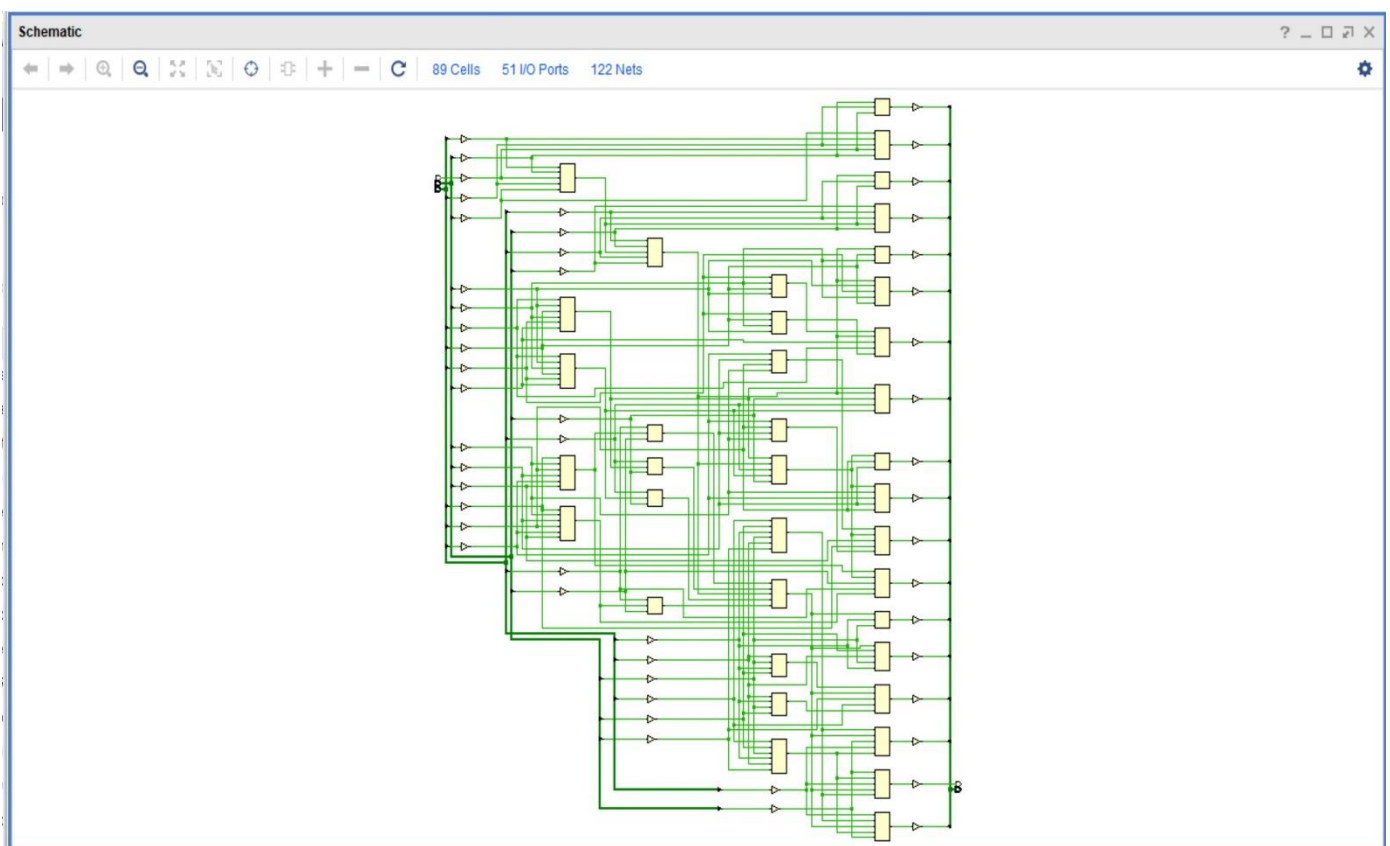


# Schematic

Eseguendo la **sintesi**, si è ottenuta una mappa (**SCHEMATIC**) che rappresenta le vere porte logiche messe a disposizione della piattaforma scelta ad inizio progetto: *ZedBoard Zynq Evaluation and Development Kit*. Quanto descritto rappresenta l'hardware.

Sono state ricavate tabelle di verità sotto forma di *LUT* (che usano *FPGA*).

Dopo aver eseguito la sintesi, si è proceduto con l'implementazione per poter fissare i collegamenti.





# SimCSA16

Per eseguire un TEST bisogna scrivere un altro file VHDL che prende il nome di **TEST BENCH**, ovvero un file dedicato alle simulazioni che non è né sintetizzabile né implementabile (nel nostro caso è chiamato **SimCSA16**).

Il **TEST BENCH** gestisce la variazione degli ingressi nel tempo.

La keyword **wait for** si utilizza per attendere un determinato tempo prima di proseguire con le istruzioni.

La keyword **others** si utilizza per assegnazioni indipendenti dalla grandezza del vettore.

```

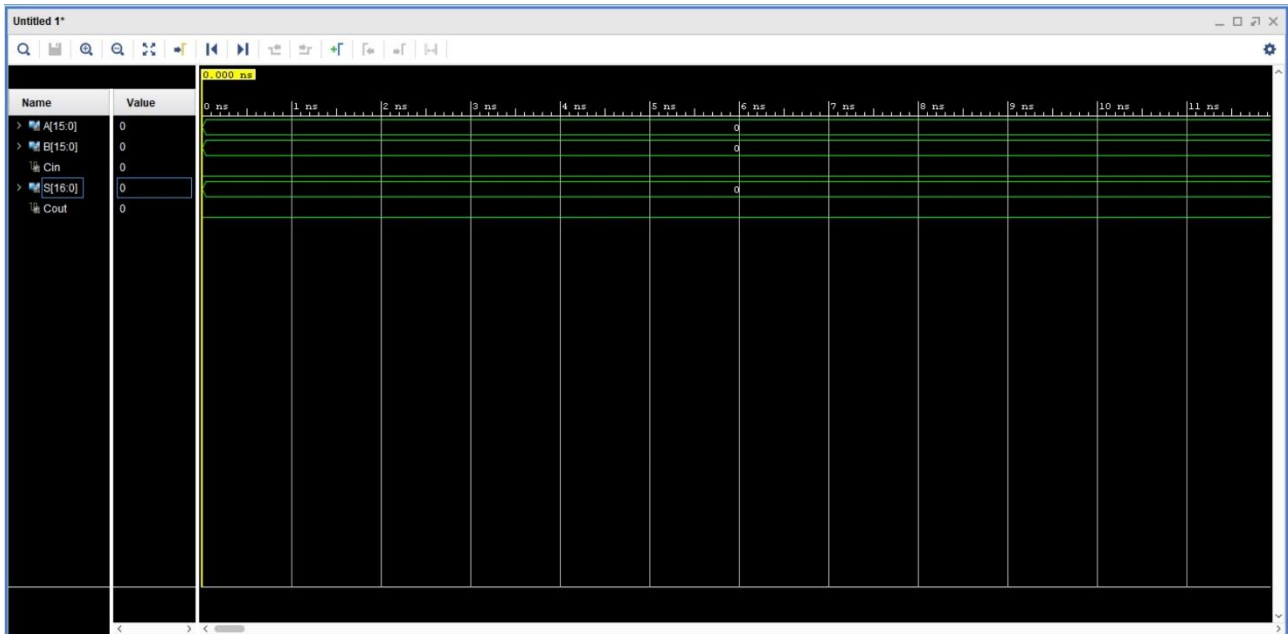
SimCSA16.vhd
C:/Users/anton/Documents/RACCOLTE/DOCUMENTI/vhdl workspace/CarrySelectAdder16bit/CarrySelectAdder16bit/srcs/sim_1/new/SimCSA16.vhd

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.all; -- ARITH = aritmetica
4  library work; --directory in cui si trova il progetto attuale
5  use work.myfunc.all; --funzione ausiliaria
6
7  entity SimCSA16 is
8  -- Port ( );
9  end SimCSA16;
10
11 architecture Behavioral of SimCSA16 is
12 component CSA16 is
13     Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
14           B : in STD_LOGIC_VECTOR (15 downto 0);
15           Cin : in STD_LOGIC;
16           S : out STD_LOGIC_VECTOR (16 downto 0);
17           Cout : out STD_LOGIC);
18 end component;
19
20 --Input
21 signal A : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
22 signal B : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
23 signal Cin : std_logic := '0';
24
25 --Output
26 signal S : STD_LOGIC_VECTOR(16 downto 0);
27 signal Cout : STD_LOGIC;
28
29 begin
30 circuito: CSA16
31     Port map( A => A,
32              B => B,
33              Cin => Cin,
34              S => S,
35              Cout => Cout);
36 process
37
38 begin
39 --t=0
40 wait for 20 ns;
41 --t=20
42
43 -- utilizzo del for loop (versione sequenziale del for), di default va è un int, quindi con l'aiuto di ARITH lo convertiamo in STD_LOGIC
44 -- da -(2^(15)) : (2^(15))-1
45 -- simulazione esaustiva (di tutti i possibili valori degli operandi)
46 for va in (-pow2(15)) to (pow2(15)-1) loop --numero di bit= 16 bit, quindi il for va da -32768 a 32767
47     A<=conv_std_logic_vector(va, 16);
48     for vb in (-pow2(15)) to (pow2(15)-1) loop
49         B<=conv_std_logic_vector(vb, 16);
50         wait for 10 ns;
51     end loop;
52 end loop;
53
54 end process;
55
56 end Behavioral;

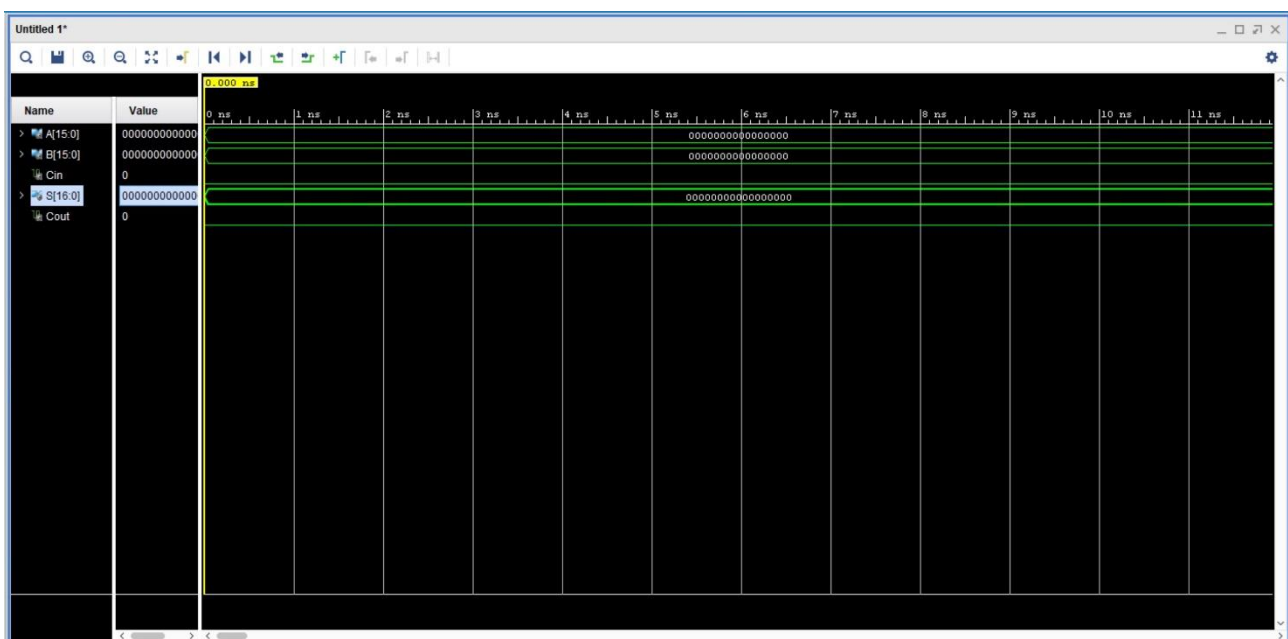
```

# SCREENSHOTS DEI TEST

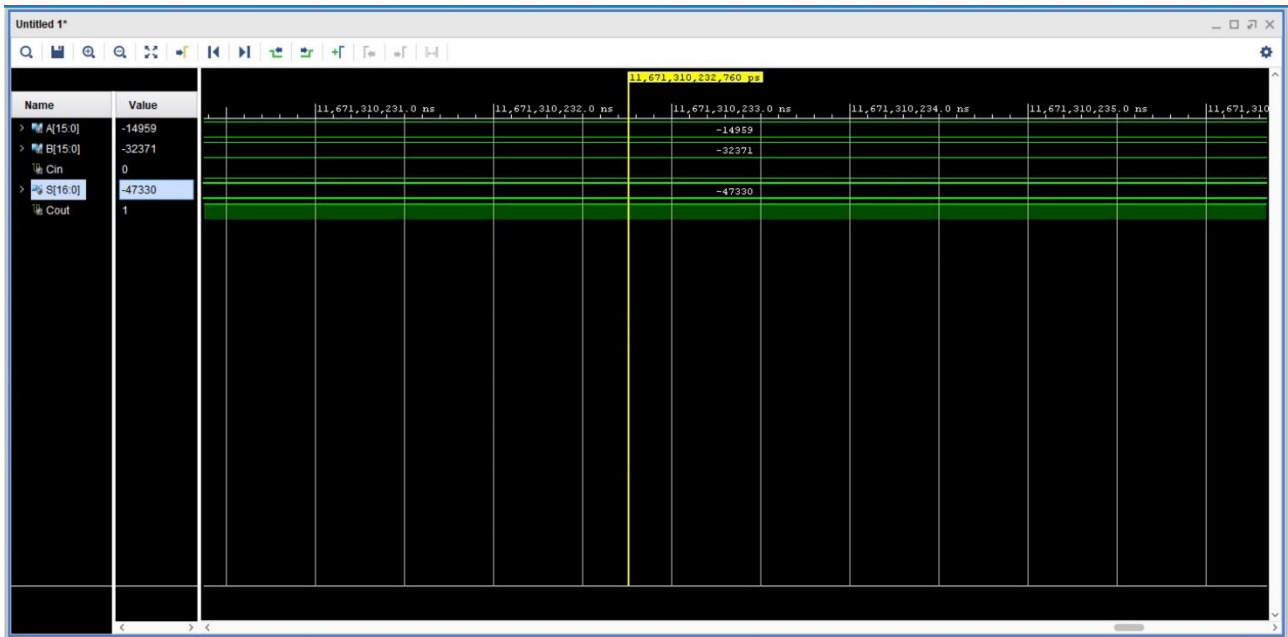
*test 0+0 in decimale*



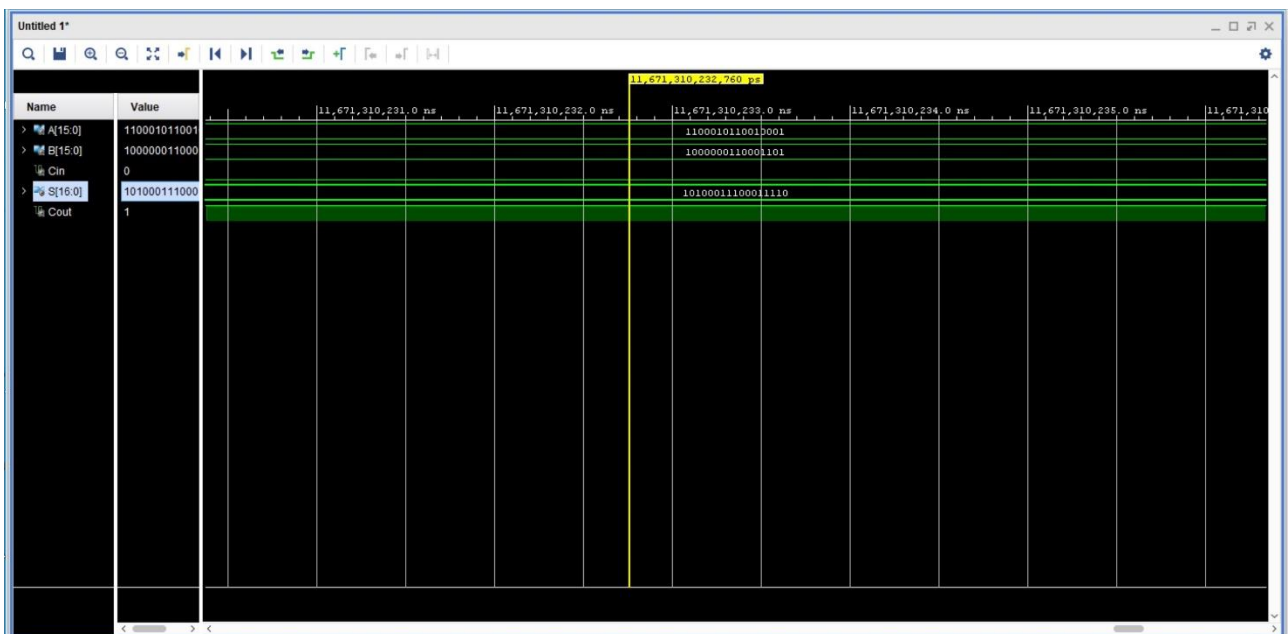
*test 0+0 binario*



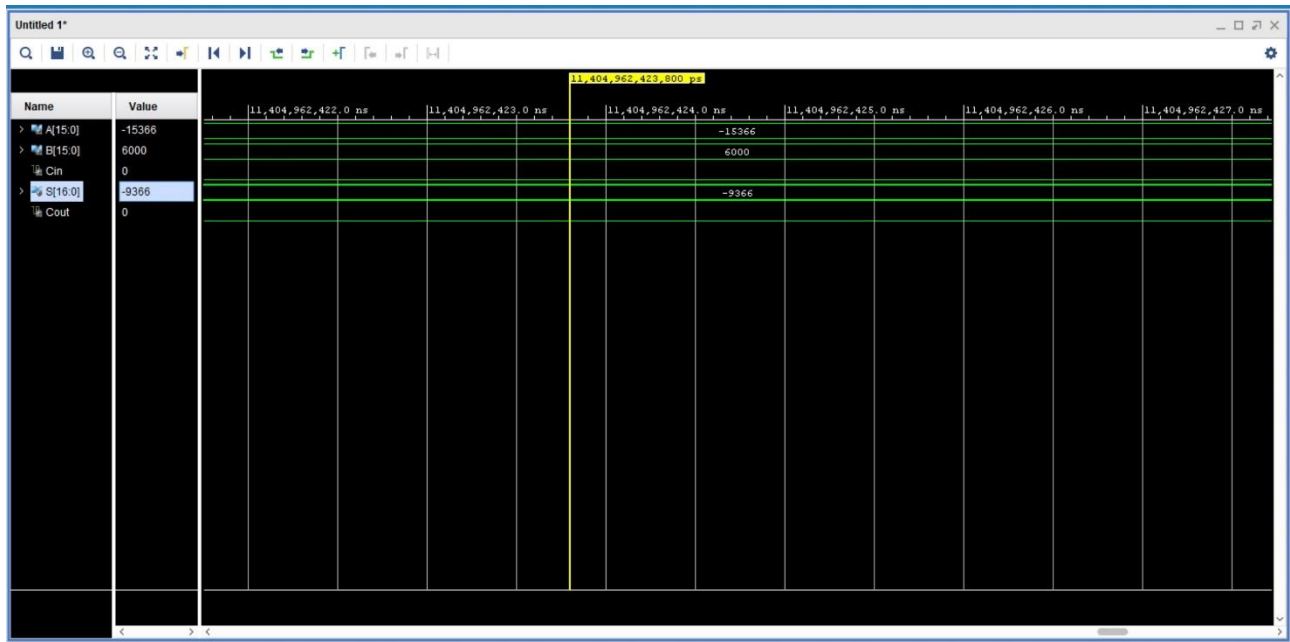
### *test $(-A) + (-B)$ in decimale*



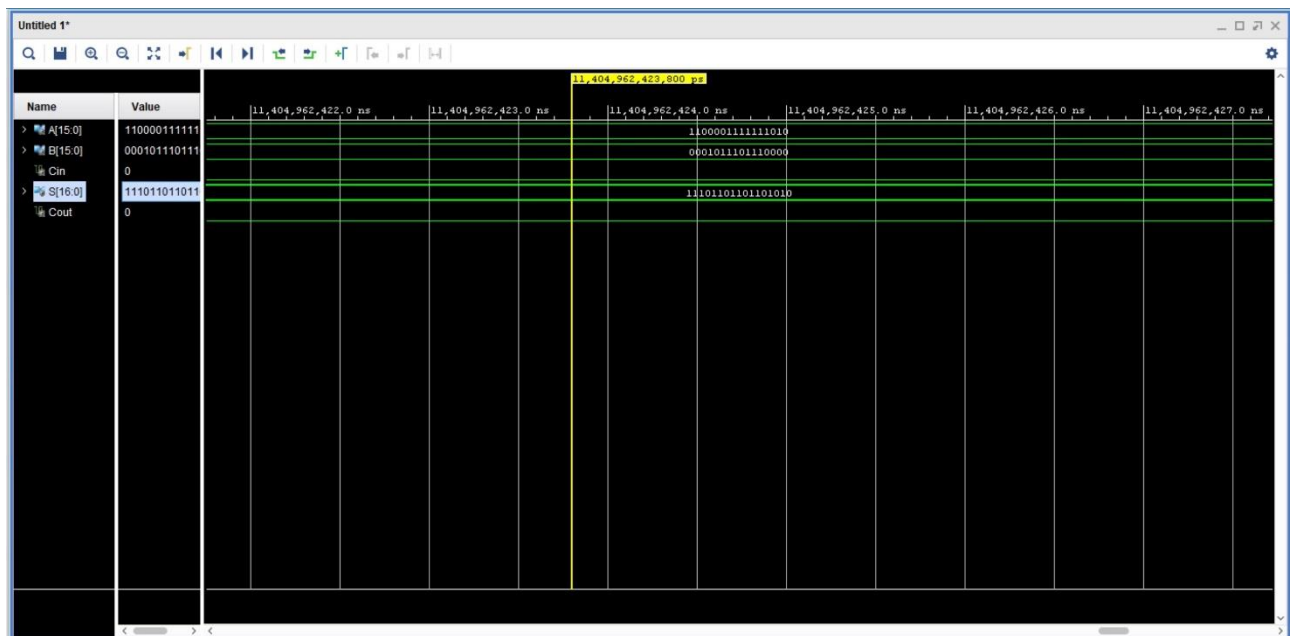
### *test $(-A) + (-B)$ in binario*

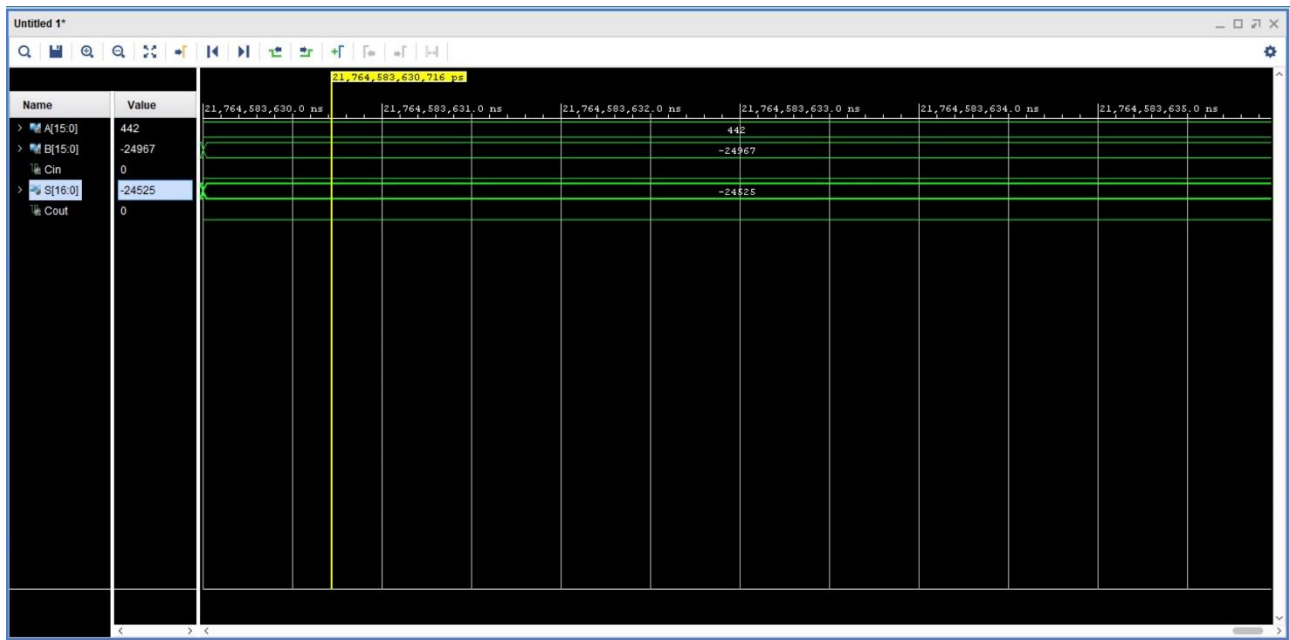
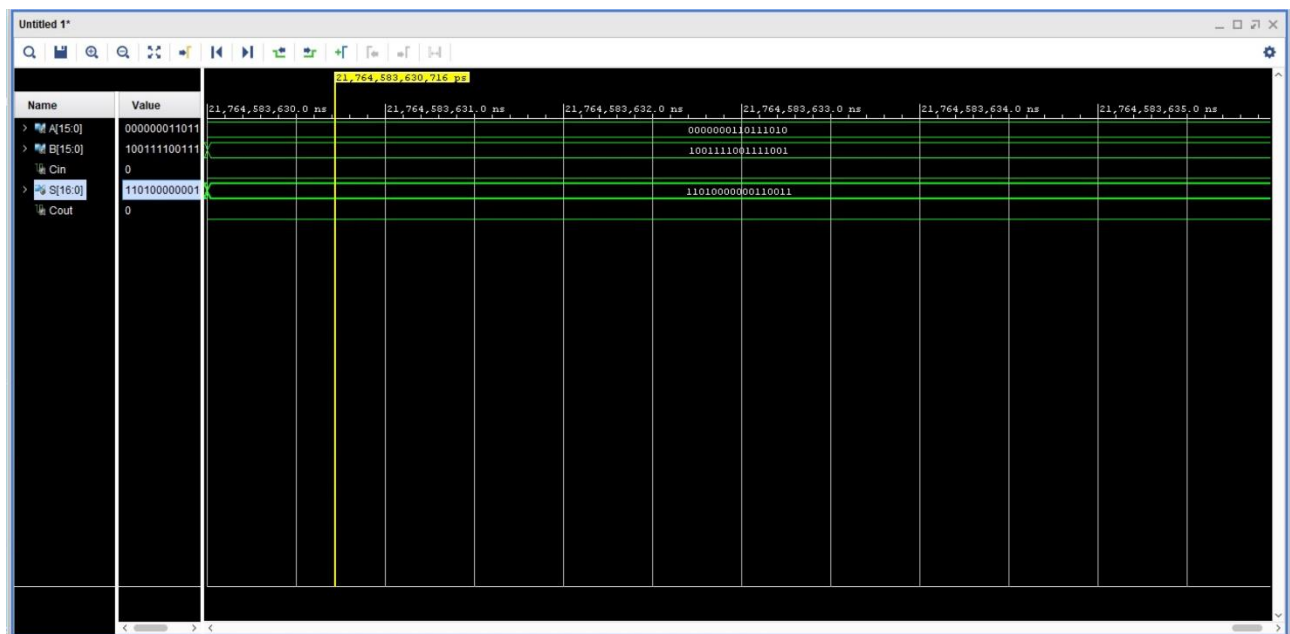


*test  $(-A) + (B)$  in decimale*

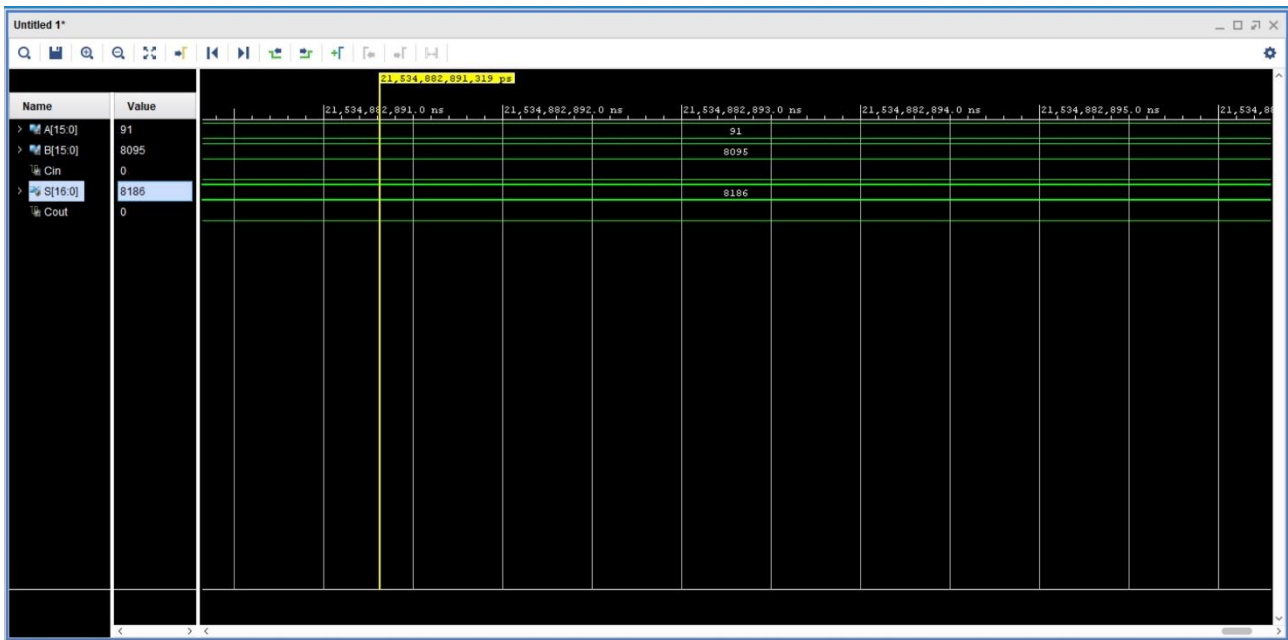


*test  $(-A) + (B)$  in binario*

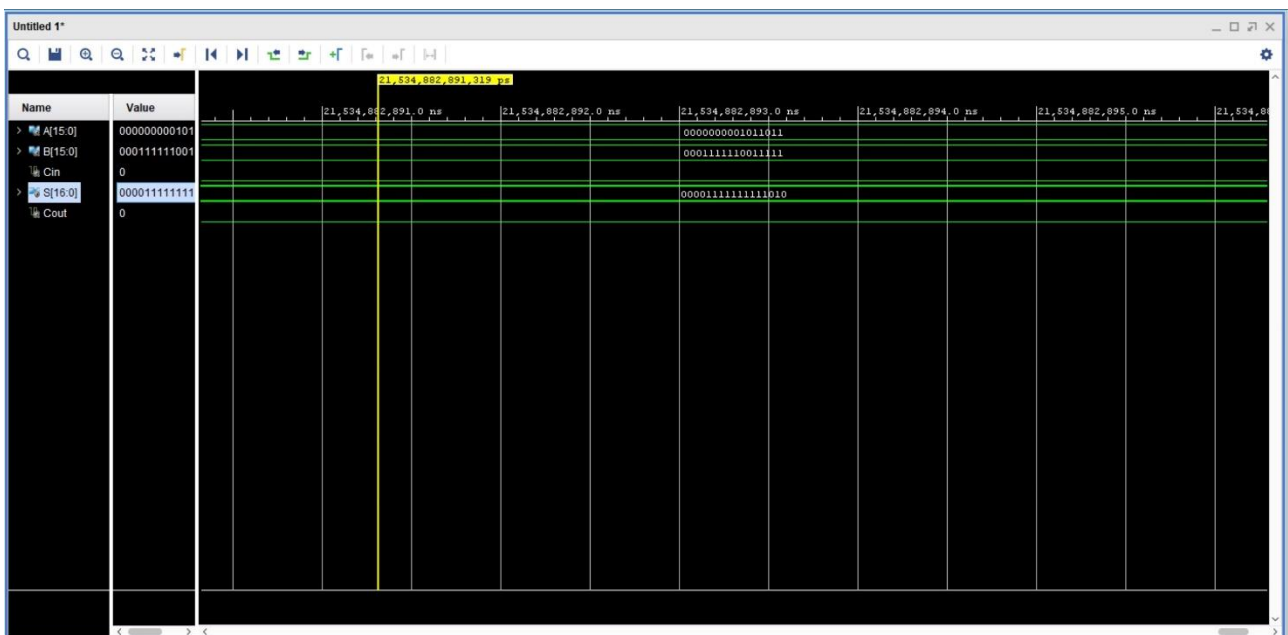


*test (A) + (-B) in decimale**test (A) + (-B) in binario*

### *test (A) + (B) in decimale*



### *test (A) + (B) in binario*



Progetto realizzato da:

**Michele Purrone**

**Antonino Vaccarella**