



CARRY-SELECT

A 32 BIT

Rabbia Marco 220216
Lazzaro Francesco 220810

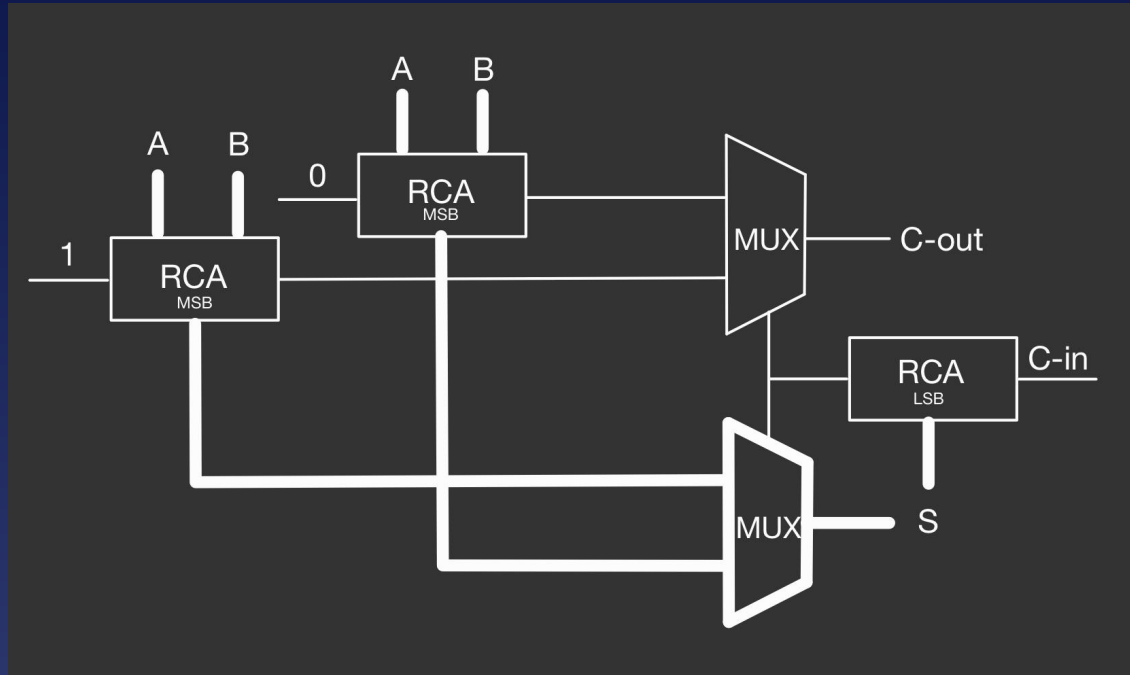
Cos'è il Carry-Select Adder

Il **Carry-Select** è un circuito sommatore che permette di velocizzare il calcolo della somma tra due operandi introducendo la ridondanza dei calcoli.

Generalmente è costituito da **Ripple carry** e **Multiplexer**. Due numeri a n-bit vengono sommati contemporaneamente attraverso i Ripple-carry, una volta assumendo come carry-input 0 e una volta 1. Dopo che i due numeri vengono calcolati, la somma corretta, così come il carry-out corretto, vengono selezionati attraverso il Mux.

Il carry select ha una struttura piuttosto semplice ma molto veloce, come vedremo nelle pagine successive.

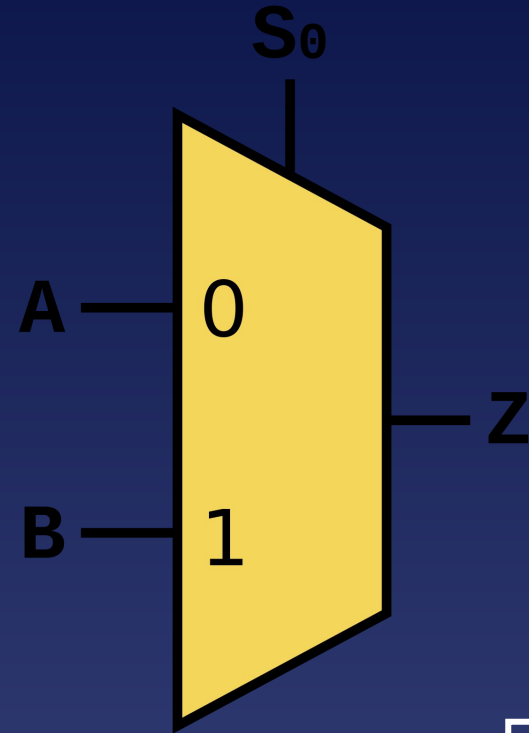
Schema di un Carry-Select



Cos'è il Multiplexer

Il **Multiplexer** è un circuito logico che consente di selezionare uno tra 2^n **ingressi** in base allo stato di “**n**” segnali di controllo. Questi ultimi individuano quale degli ingressi bisogna selezionare per poi incanalarli verso l'unica uscita del circuito.

Schema



Codice VHDL Multiplexer

```
entity MUX is
    Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          s : in STD_LOGIC;
          cout : out STD_LOGIC);
end MUX;

architecture myMUX of MUX is
begin
    cout <= (a and (not s)) or (b and s);
end myMUX;
```

Vengono utilizzati dei dati di tipo “**std_logic**”, che fanno parte della libreria **IEEE**, in modo tale da non creare conflitti tra i segnali ed evitare che essi assumano solo i valori di semplici bit 0 o 1. Altre possibili assegnazione di valori std_logic:

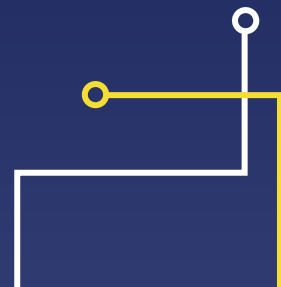
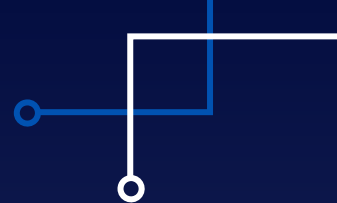
- “**U**” : non inizializzato
- “**X**” : sconosciuto
- “**Z**” : alta impedenza
- “**L**” : segnale debole (vicino allo zero)
- “**H**” : segnale alto (vicino ad 1)
- “**-**” : non ha importanza

Cos'è il Full-Adder

Un **Full-Adder** è un circuito logico caratterizzato da **tre ingressi** e **due uscite**. La sua funzionalità è quella di eseguire la **somma di due numeri** espressi in bit e con la capacità di considerare un riporto.

Esso si compone a sua volta da due sommatori (Half-Adder) che però non sono in grado di mantenere il riporto.

Per sommare un eventuale riporto è possibile inserire due Half-Adder in cascata



Codice VHDL Full-Adder

```
entity FA is
Port ( a : in std_logic;
      b : in std_logic;
      cin : in std_logic;
      cout : out std_logic;
      s : out std_logic);
end FA;

architecture myFA of FA is
signal p :std_logic;
signal g :std_logic;
begin
    p <= a xor b;
    g <= a and b;
    cout <= g OR (p and cin);
    s <= p xor cin;
end myFA;
```

Un'assegnazione di un segnale equivale a costruire un “driver” (ovvero un'uscita di una porta logica) per il segnale. Si usa il comando “**signal**” e vengono utilizzati per rappresentare collegamenti nel circuito che non sono dunque riconducibili ad ingressi o uscite ma trasportano un segnale da una porta all'altra.

Schema di un Full-Adder

Elettronica digitale: Half Adder and Full Adder

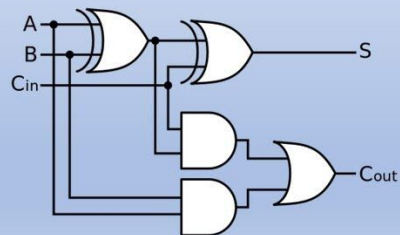
Half Adder

| A | B | Somma | Riporto |
|---|---|-------|---------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



Full Adder

| A | B | Riporto Preced | Somma | Riporto |
|---|---|-------------------|-------|---------|
| 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 |






Cos'è il Ripple-Carry Adder

Il **Ripple-Carry adder** o addizionatore a propagazione di riporto è un circuito costituito al suo interno da più sommatore (Full Adder) collegati a cascata in modo che il riporto di un sommatore sia il riporto in ingresso del successivo.

Esso non fa nient'altro che calcolare la somma così come verrebbe calcolata a mano. La sua architettura è semplice, ma anche lenta. Uno dei punti più importanti da considerare in questo carry adder è che per visualizzare l'output finale bisogna tener conto di un ritardo intrinseco, perché ogni Full Adder per generare il proprio output deve aspettare il completamento del sommatore precedente.

Quindi ci sarà un ritardo che cresce in base al numero di porte che il segnale deve attraversare che è possibile calcolare tramite il **CRITICAL PATH** (percorso critico) ovvero il percorso con il maggior numero di porte nella quali passa il segnale.



Codice VHDL RCA

```
entity RCA16b is
    Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
          b : in STD_LOGIC_VECTOR (15 downto 0);
          cin : in STD_LOGIC;
          cout: out STD_LOGIC;
          s : out STD_LOGIC_VECTOR (15 downto 0));
end RCA16b;

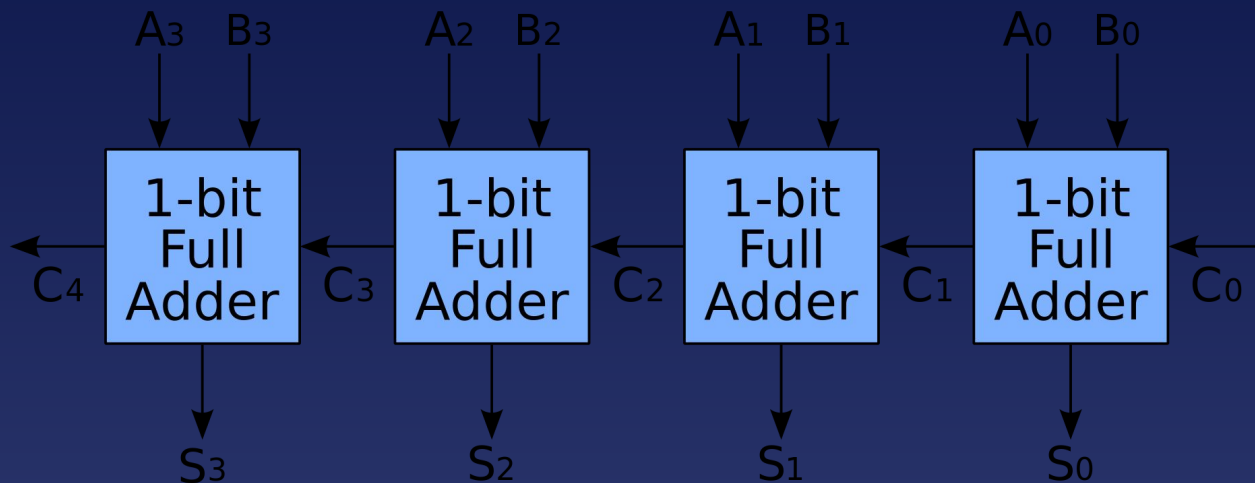
architecture myRCA of RCA16b is
    signal exta , extb : STD_LOGIC_VECTOR (16 downto 0);
    signal C: std_logic_vector (17 downto 0);
    component FA
        port(a,b, cin: in std_logic;
             s,cout: out std_logic);
    end component;
begin
    exta<= a(15) & a;
    extb<= b(15) & b;
    C(0) <= cin;

    FAi: for i in 0 to 15 generate
        myFA: FA port map(a => exta(i),
                        b => extb(i),
                        cin => C(i),
                        cout => C(i+1),
                        s=> s(i));
    end generate;
    cout <= C(16);
end myRCA;
```

In questo codice vhdl si fa uso di due importanti comandi quali il **“for generate”**, che serve ad istanziare un numero finito (dato dal range inserito) di componenti uguali, e l’istruzione **“port map”** fa parte dell’istanziamento dei componenti, in cui si dichiara quali segnali locali devono essere collegati alle porte di ingresso e/o uscita del circuito.

Schema di un Ripple-Carry

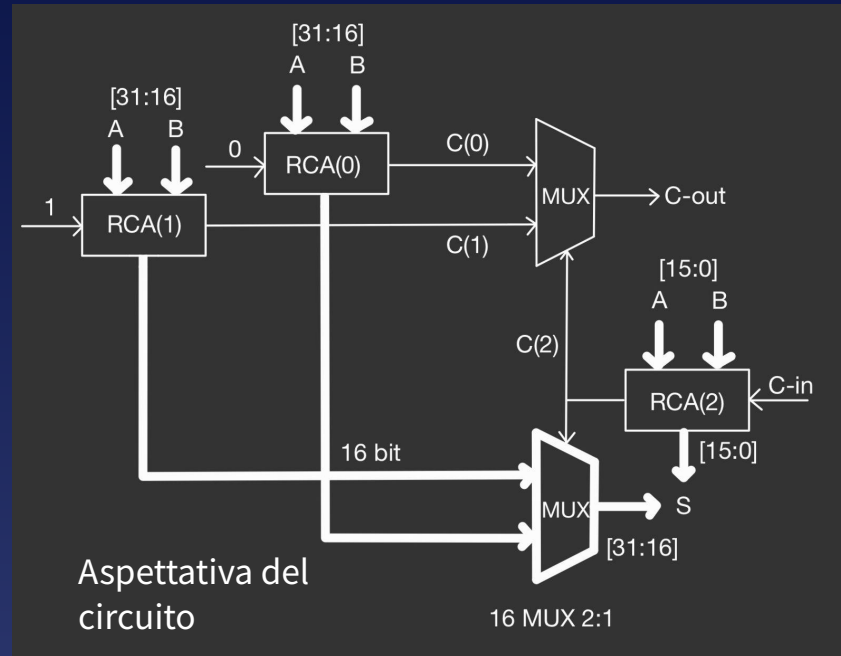
Esempio di Ripple-Carry a 4 bit:



Carry-Select a 32 bit

Per la progettazione del CSA32b abbiamo utilizzato 3 Report-Carry a 16 bit:

- 1 **LSB** (myRCA2) in cui vengono sommati i primi 16 bit meno significativi (da 0 a 15), riempiendo direttamente le prime 16 posizioni del vettore S, e che genera il segnale Cout, salvato nel segnale C(2), che diventerà poi il bit di controllo per i mux successivi.
- 2 **MSB** (myRCA0 e myRCA1) in cui vengono sommati parallelamente gli altri bit più significativi, una volta con '0' come Cin e una volta '1'. Ogni coppia di bit viene poi selezionata attraverso un mux che utilizza il segnale C(2) come bit di controllo. I due segnali Cout dei due RCA MSb vengono selezionati sempre attraverso un mux.



Codice VHDL CSA

```
entity CSA32b is
    Port ( A : in STD_LOGIC_VECTOR (31 downto 0);
          B : in STD_LOGIC_VECTOR (31 downto 0);
          Cin : in STD_LOGIC;
          Cout : out STD_LOGIC;
          S : inout STD_LOGIC_VECTOR (31 downto 0));
end CSA32b;

architecture myCSA32b of CSA32b is

    signal sLSB, sMSB0, sMSB1, aLSB, bLSB, aMSB, bMSB: std_logic_vector (15 downto 0);
    signal C: std_logic_vector (2 downto 0);

    component RCA16b
        Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
              b : in STD_LOGIC_VECTOR (15 downto 0);
              cin : in STD_LOGIC;
              cout: out STD_LOGIC;
              s : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component Mux
        Port ( a : in STD_LOGIC;
              b : in STD_LOGIC;
              s : in STD_LOGIC;
              cout : out STD_LOGIC);
    end component;
```

```
begin

    aLSB <= A(15 downto 0);
    bLSB <= B(15 downto 0);
    aMSB <= A(31 downto 16);
    bMSB <= B(31 downto 16);

    myRCA2: RCA16b port map (a => aLSB,
                             b => bLSB,
                             cin => Cin,
                             cout => C(2),
                             s => sLSB);

    myRCA1: RCA16b port map (a => aMSB,
                             b => bMSB,
                             cin => '1',
                             cout => C(1),
                             s => sMSB1);

    myRCA0: RCA16b port map (a => aMSB,
                             b => bMSB,
                             cin => '0',
                             cout => C(0),
                             s => sMSB0);

    myMux: for i in 0 to 15 generate
        MUXi: Mux port map (a => sMSB0(i),
                             b => sMSB1(i),
                             s => C(2),
                             cout => S(i+16));
    end generate;

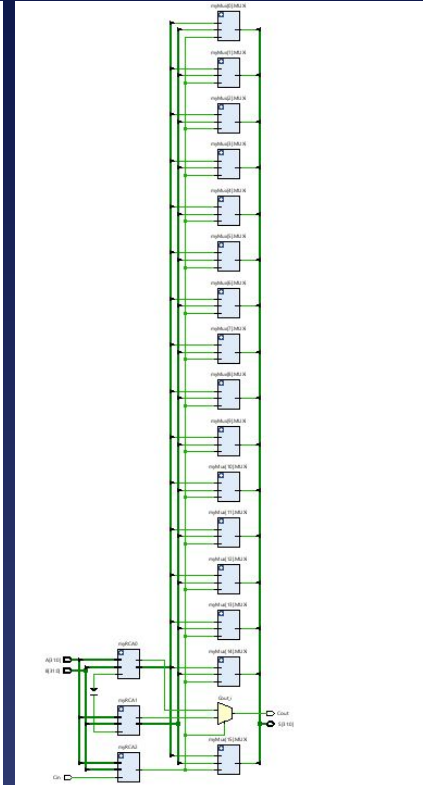
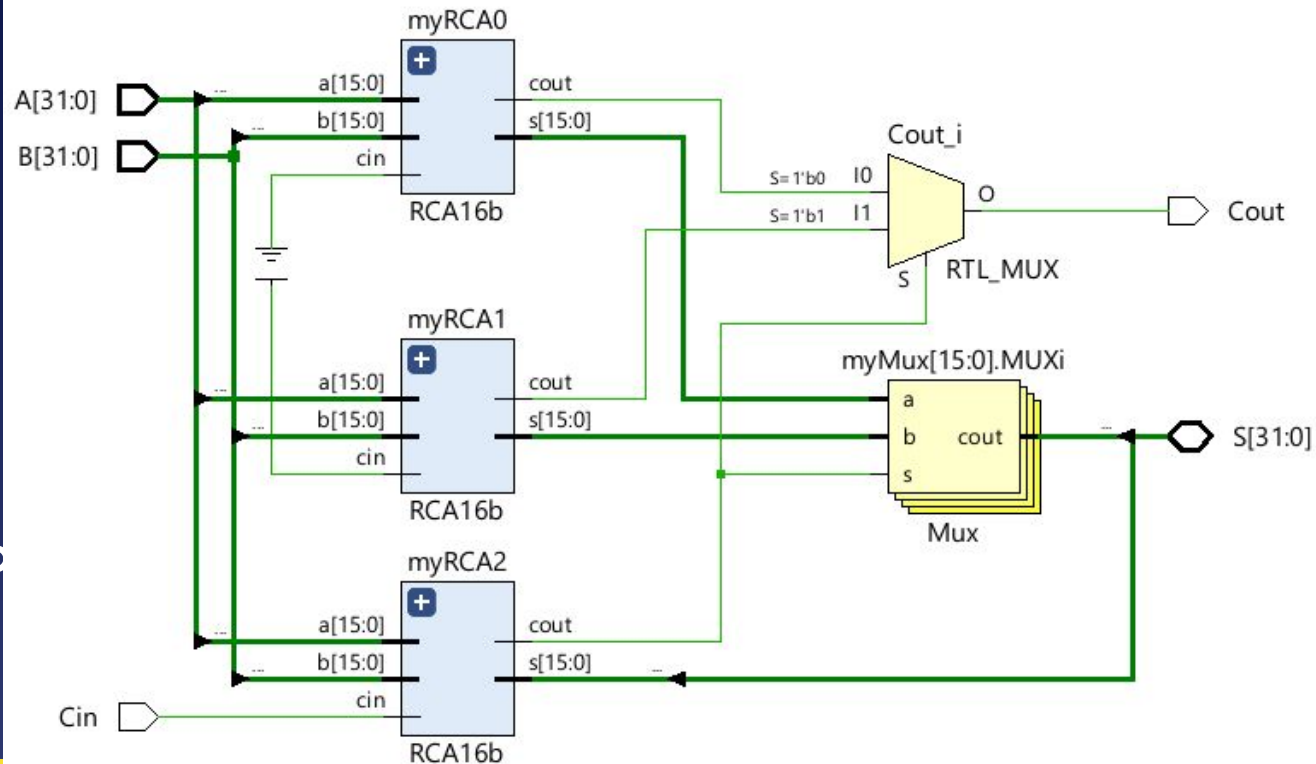
    with C(2) select
        Cout <=  C(0) when '0',
                  C(1) when '1',
                  'X' when others;

    S(15 downto 0) <= sLSB(15 downto 0);

end myCSA32b;
```

Oltre ad usare tutti i comandi precedentemente citati, si utilizza un'assegnazione condizionale della forma **with x select**.

Schematic



Testbench

Una **testbench** è un file VHDL che implementa una simulazione di un circuito in cui vengono definiti i valori dei segnali in ingresso. Non sono presenti elementi di input o output ma vengono utilizzati dei segnali che conterranno i valori della simulazione.

La testbench non viene sintetizzata ma risulta essere utile per valutare la correttezza del codice.

L'opzione ottimale sarebbe effettuare una simulazione esaustiva ovvero un test in cui si inviano in input tutte le possibili combinazioni di operandi. Nel caso di numeri a 32 bit il range risulta essere troppo ampio per una simulazione da effettuare in tempi ragionevoli, dunque la simulazione è stata effettuata in un range stabilito.

Per generare le varie combinazioni si può fare uso del costrutto for loop che assegna alle variabili di input un valore per volta.

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

entity SIM_CSA32b is
end SIM_CSA32b;

architecture mySIM of SIM_CSA32b is
  component CSA32b
    Port (A : in STD_LOGIC_VECTOR (31 downto 0);
          B : in STD_LOGIC_VECTOR (31 downto 0);
          Cin : in STD_LOGIC;
          Cout : out STD_LOGIC;
          S : inout STD_LOGIC_VECTOR (31 downto 0));
  end component;

  signal Ia, Ib, Os: std_logic_vector (31 downto 0);
  signal Icin, Ocout: std_logic;

begin

  CUT: CSA32b port map (Ia, Ib, Icin, Ocout, Os);
  Icin <= '0';

  Sym: process begin
    for i in 100 to 128 loop
      Ia <= conv_std_logic_vector(i,32);
      for j in 90 to 118 loop
        Ib <= conv_std_logic_vector(j,32);
        wait for 20 ns;
      end loop;
    end loop;
  end process;

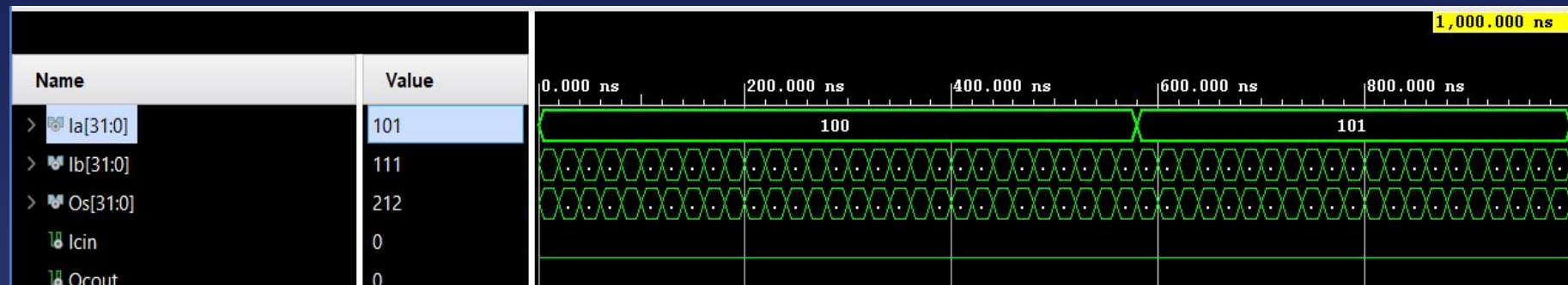
end mySIM;
```

Simulazione behavioral

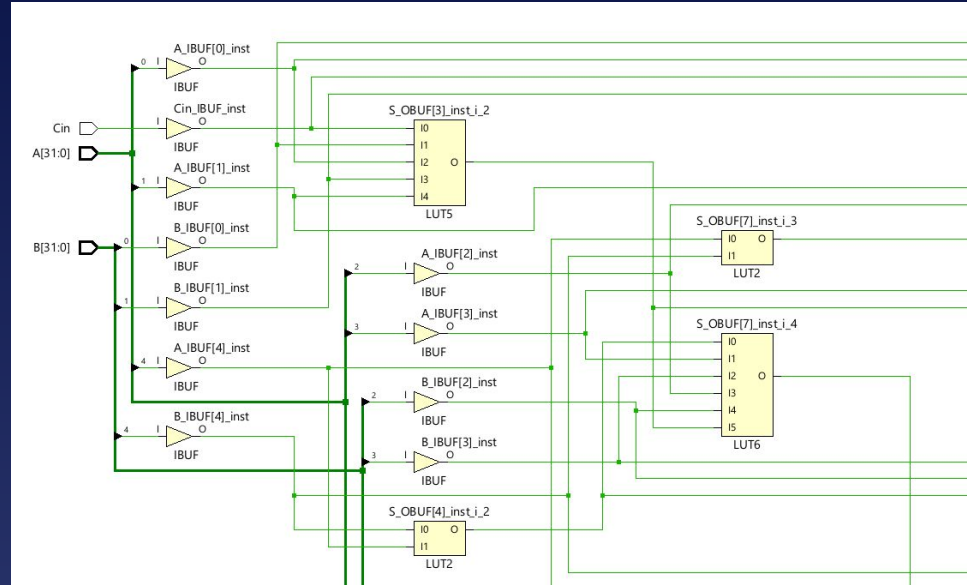
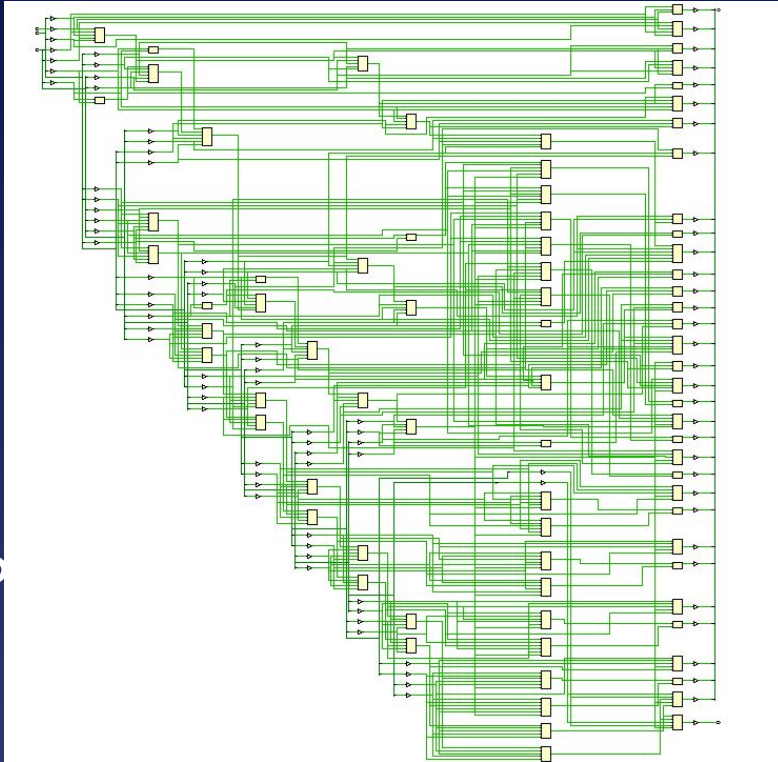
La simulazione behavioral non tiene conto degli aspetti reali di un circuito, dunque all'inizio della simulazione non è presente alcun ritardo nel calcolo della somma corrente.

In particolare dopo 1000 ns la somma risulta essere **212** in decimale.

Attraverso una **post-synthesis timing simulation** è possibile osservare un comportamento reale del circuito.

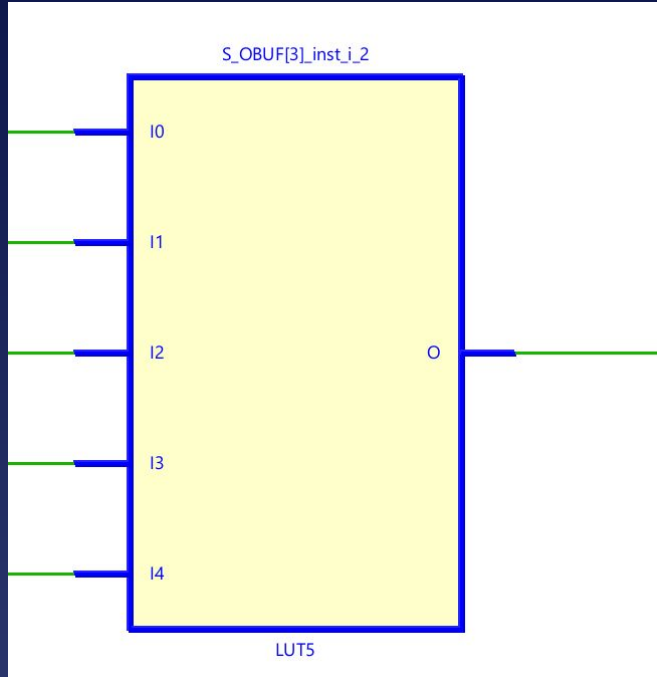


Schematic Post-Synthesis



Durante la sintesi il circuito logico iniziale viene convertito in uno reale mediante l'utilizzo delle **LUT**.

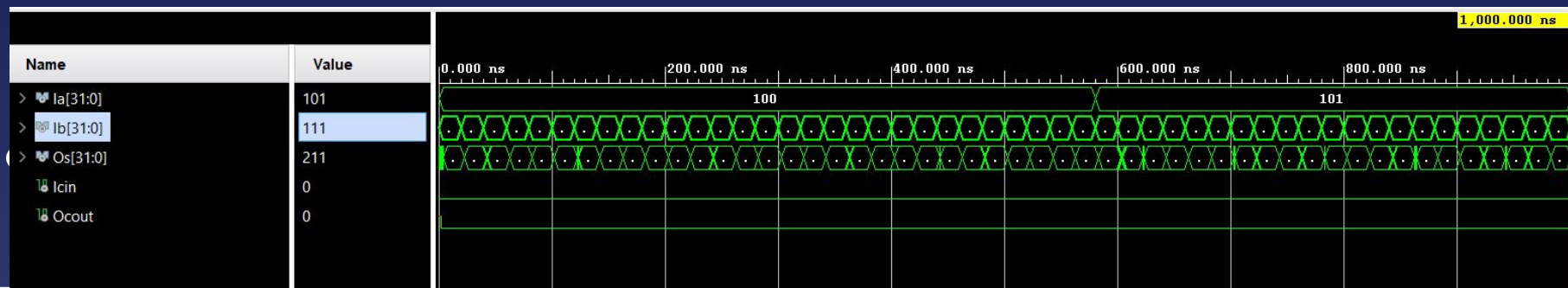
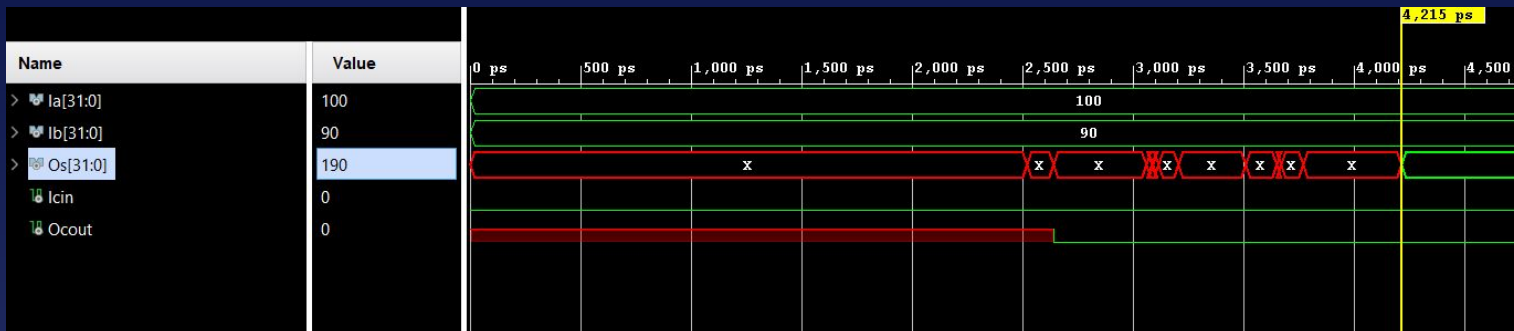
Cosa sono le LUT



Per **LUT** (**Look Up Table**) si intende una struttura dati, usata per sostituire operazioni di calcolo a runtime con una più semplice operazione di consultazione. Il guadagno di velocità può essere significativo, poiché recuperare un valore dalla memoria è spesso più veloce che sottoporre a calcoli con tempi di esecuzione dispendiosi. Il calcolo della funzione richiede solo una singola ricerca nella memoria indipendentemente dalla complessità della funzione. L'indirizzo è l'input della funzione e il valore a quell'indirizzo è l'output della funzione. Lo svantaggio è che richiede memoria, soprattutto se è necessaria un'alta risoluzione per l'input della funzione.

| I4 | I3 | I2 | I1 | I0 | O=I0 & I1 & I3 + I1 & I3 & I4 + I1 & I2 & I3 + I0 & I1 & I3 & I4 + I0 & I2 & I3 + I0 & I2 & I3 & I4 + I1 & I2 & I3 & I4 + I2 & I3 & I4 |
|----|----|----|----|----|--|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |

Simulazione Post-Synthesis



Confronto simulazione Behavioral e Post-Synthesis

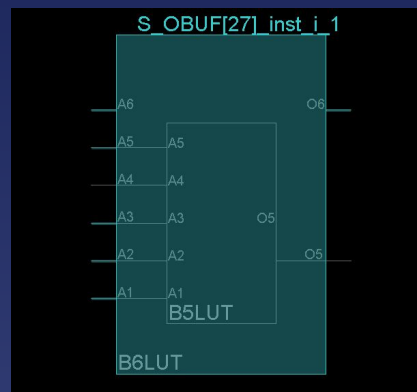
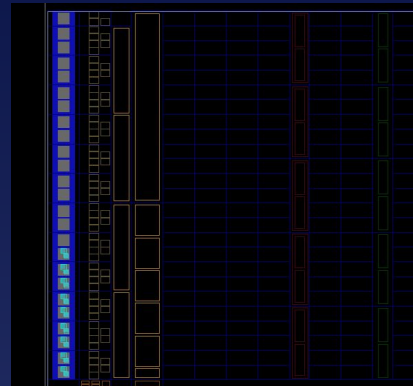
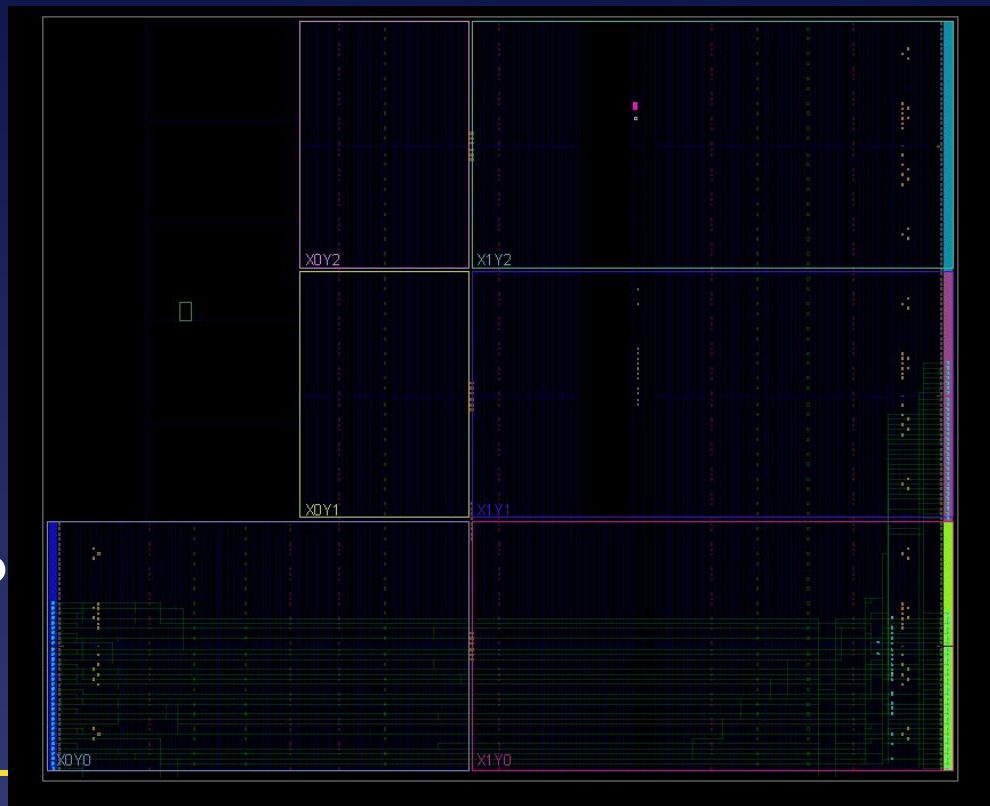
Come si può notare nella slide precedente, dopo lo stesso intervallo di campionamento di esattamente 1000 ns la somma risulta essere **211**. Questa differenza rispetto alla simulazione behavioral è dovuta ad un ritardo iniziale che riguarda l'elaborazione degli ingressi e che genera quindi sul segnale **Os** la non specificazione 'X'.

In particolare l'istanziamento della prima somma avviene dopo **4,215 ns**.

La sintesi seppur rappresenta un primo comportamento reale del circuito, questa non è ottimizzata, come risultato si ottengono dei tempi di esecuzione non ottimali.

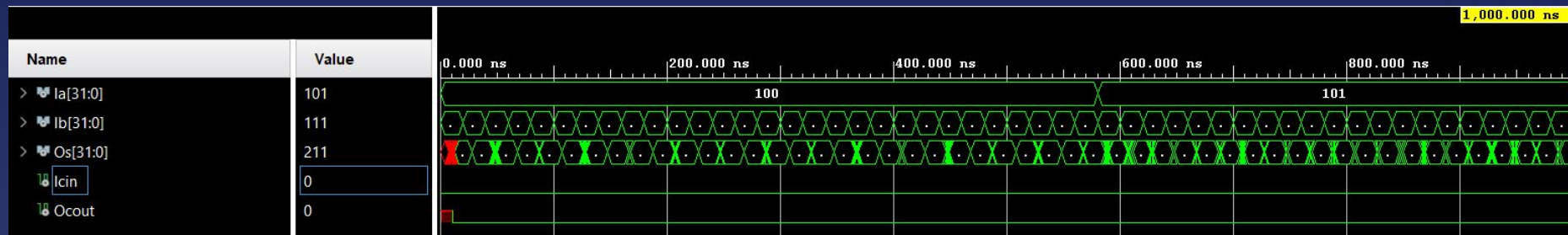
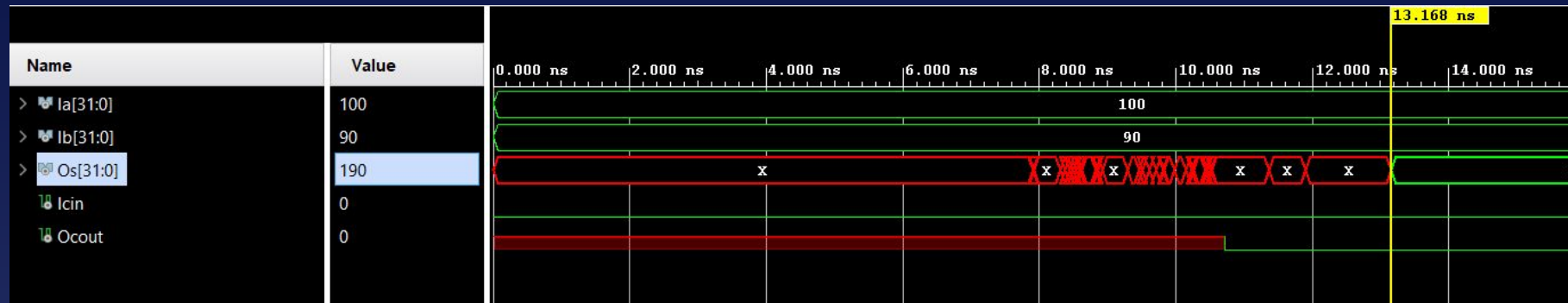
E' importante dunque passare ad una **implementazione del codice** ed a una **simulazione post-implementation**.

Post-Implementation Design



Così è possibile visualizzare quali componenti del device a nostra disposizione sono utilizzati dal circuito.

Simulazione Post-Implementation

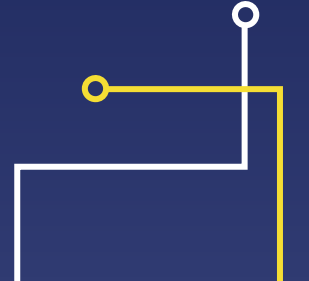





Confronto simulazione Post-Synthesis e Post-Implementation

Confrontando le due timing simulation, dopo i 1000 ns la somma finale risulta uguale, tuttavia si può notare un'evidente discrepanza nel ritardo iniziale che risulta essere **maggiore** nella **post-Implementation**, ovvero **13,128 ns**.

Questo aumento del ritardo è dovuto all'implementazione su device, infatti questa simulazione rappresenta il comportamento reale di un Carry-Select a 32 bit che opera all'interno di un chip.



Report

Completando le simulazioni si ottengono dei **File Report** che contengono delle descrizioni sui consumi ed efficienza del circuito.

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: **24.794 W (Junction temp exceeded!)**

Design Power Budget: **Not Specified**

Power Budget Margin: **N/A**

Junction Temperature: **125,0°C**

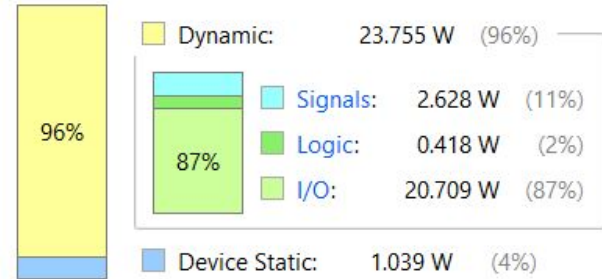
Thermal Margin: **-226,0°C (-18,8 W)**

Effective θ_{JA} : 11,5°C/W

Power supplied to off-chip devices: 0 W

Confidence level: **Low**

On-Chip Power



| Site Type | Used | Fixed | Prohibited | Available | Util% |
|-----------------------|------|-------|------------|-----------|-------|
| Slice LUTs | 69 | 0 | 0 | 53200 | 0.13 |
| LUT as Logic | 69 | 0 | 0 | 53200 | 0.13 |
| LUT as Memory | 0 | 0 | 0 | 17400 | 0.00 |
| Slice Registers | 0 | 0 | 0 | 106400 | 0.00 |
| Register as Flip Flop | 0 | 0 | 0 | 106400 | 0.00 |
| Register as Latch | 0 | 0 | 0 | 106400 | 0.00 |
| F7 Muxes | 0 | 0 | 0 | 26600 | 0.00 |
| F8 Muxes | 0 | 0 | 0 | 13300 | 0.00 |

* Warning! LUT value is adjusted to account for LUT combining.

| Ref Name | Used | Functional Category |
|----------|------|---------------------|
| IBUF | 65 | IO |
| OBUF | 33 | IO |
| LUT5 | 33 | LUT |
| LUT6 | 22 | LUT |
| LUT2 | 16 | LUT |
| LUT3 | 10 | LUT |

Per questo progetto sono state utilizzate in totale:

- 69 LUT
- 98 Porte I/O