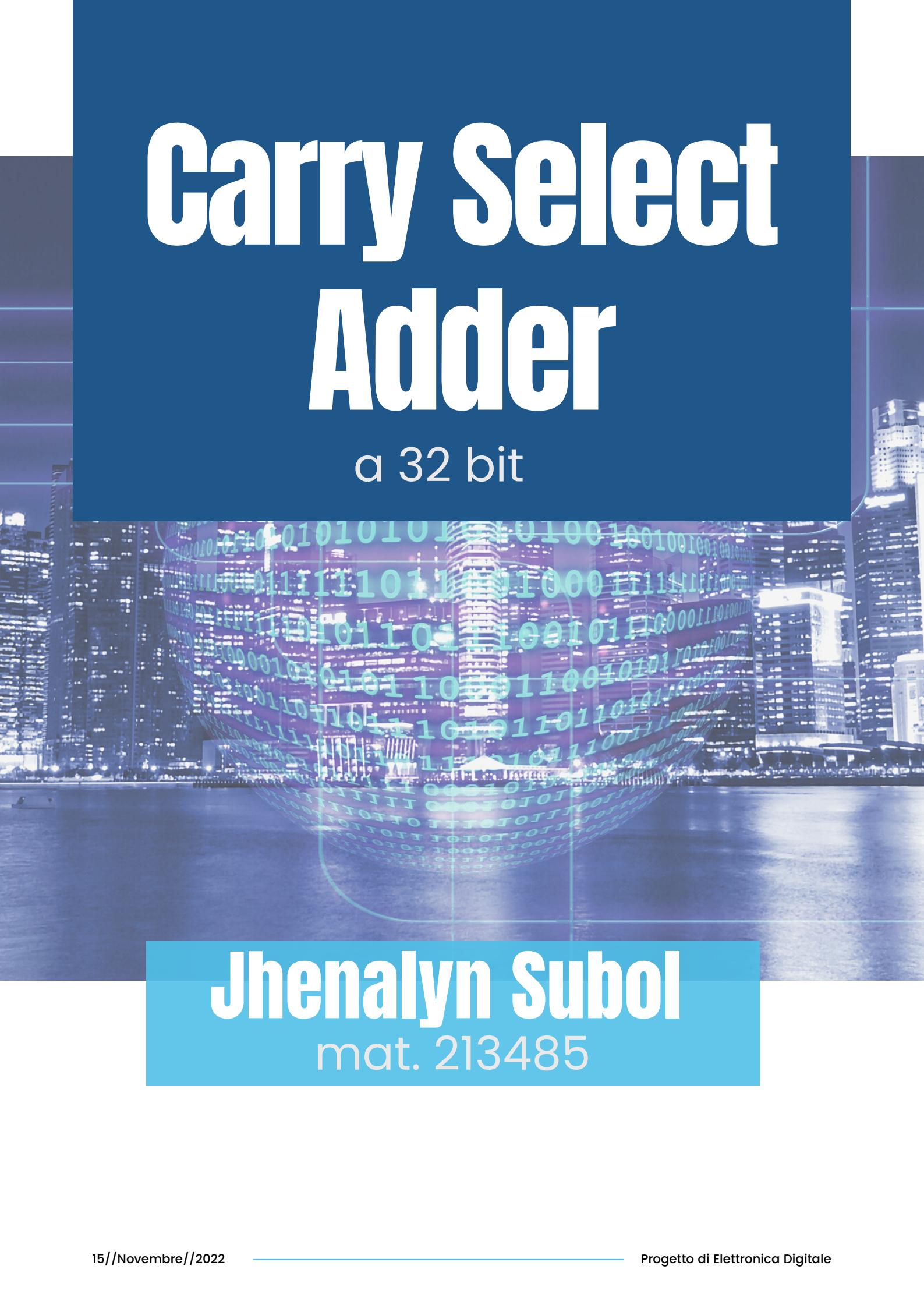


# Carry Select Adder

a 32 bit



Jhenalyn Subol  
mat. 213485

# Indice

**01 Cos'è il Sommatore Carry Select**

**02 Traccia del progetto**

**03 Schema a blocchi**

**04 Specifiche CSA a 32 bit**

**05 VHDL Ripple Carry, Multiplexer,  
Carry Select a 8 bit, Carry Select  
a 32 bit**

**06 Test Bench**

**07 Caratterizzazione del circuito**

# Cos'è il Sommatore Carry-Select

## Funzionamento generale

Il carry-select adder è un particolare circuito addizionatore il cui vantaggio è l'elevata velocità di calcolo rispetto ad un comune Ripple Carry Adder(RCA), a dispetto però di un più alto costo al livello di porte logiche usate.

Un CSA ad  $n$  bit è generalmente costituito da un RCA ad  $n/2$  bit che effettua la somma degli  $n/2$  bit meno significativi, mentre per gli  $n/2$  bit più significativi, la somma viene eseguita nel seguente modo:

- Si introducono 2 RCA da  $n/2$  bit con il riporto in entrata fissato a '0' mentre l'altro ad '1'.
- Le somme di ogni bit entrano in un Multiplexer, che ne seleziona la somma finale con il segnale di controllo preso dal riporto in uscita del RCA di bit meno significativi.

Questo calcolo simultaneo della somma degli  $n/2$  bit più significativi rende elevata la velocità di calcolo in quanto non c'è bisogno di propagare il riporto fino all'ultimo Full Adder del RCA. Infatti, nel secondo blocco, i 2 RCA effettuano a priori la somma dei bit più significativi e si sceglie la somma corretta fra i due RCA in base al riporto in uscita del primo blocco.

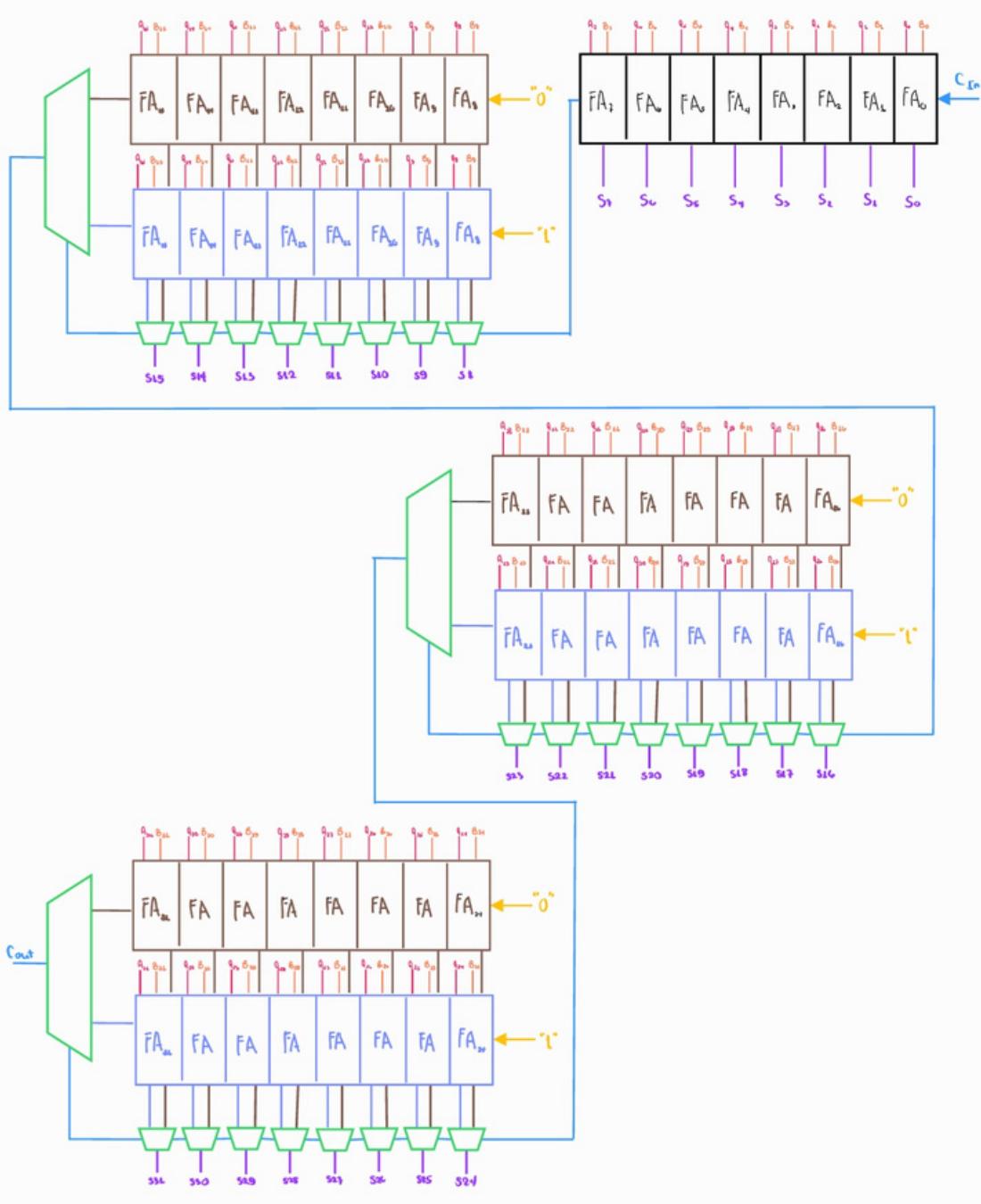
# Traccia del progetto

Si richiede di progettare un sommatore CARRY-SELECT a 32-bit, assumendo che gli ingressi siano UNSIGNED.

Bisogna preparare una relazione da consegnare in formato PDF in cui vanno riportati: i codici VHDL(circuiti e testbench), screenshots delle simulazioni e caratterizzazione in termini di risorse occupate.

# Schema a blocchi di CSA a 32 bit

Proposed CSA implementation:



# Specifiche Carry Select Adder

Un Carry Select Adder si compone generalmente da blocchi di ripple-carry a sua volta composta da blocchi di Full-Adder. Ogni Full-Adder è capace di sommare due bit per volta. Dunque associando n Full Adder in modo da propagare il riporto fra un componente e il successivo, il ripple-carry potrà calcolare la somma di numeri composti da n bit.

Per poter costruire il CSA a 32 bit, ho pensato di dividere l'intero blocco a 4 blocchi di RCA ognuno a 8 bit. Il primo blocco effettua la somma di primi 8 bit meno significativi. Mentre il 2°, 3°, e 4° blocchi sono costruiti secondo la definizione di un CSA. Quindi, ognuno di questi blocchi avrà:

- un RCA che riceve come carry-in il valore '0' ;
- un altro RCA che riceve un carry-in pari a '1' ;
- 8 multiplexer che scelgono le somme corrette tra i due RCA ricevendo come carry-in il carry-out del 1° blocco
- 1 multiplexer gestisce la scelta fra i risultati dei due RCA in base al riporto ottenuto dal blocco precedente.

Invece, non occorre gestire un eventuale overflow di questa operazione in quanto stiamo considerando i tipi di dati senza segno (unsigned).

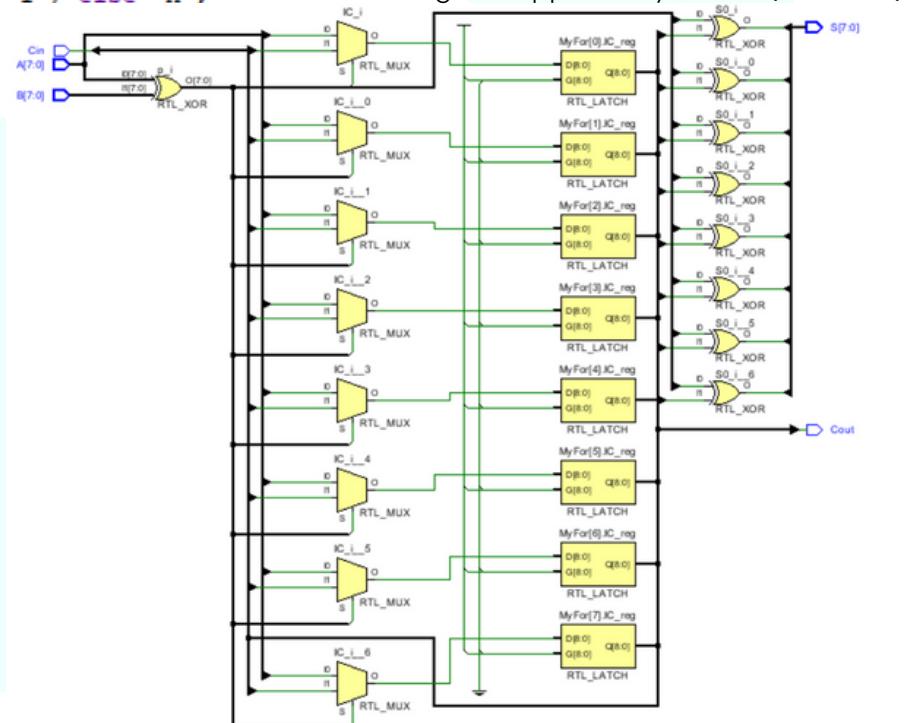
# Codice VHDL

## Ripple – Carry Adder

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity RippleCarry is
5     generic(nb:integer:=8);
6     Port ( A : in STD_LOGIC_VECTOR (nb-1 downto 0);
7             B : in STD_LOGIC_VECTOR (nb-1 downto 0);
8             Cin : in STD_LOGIC;
9             S : out STD_LOGIC_VECTOR (nb-1 downto 0);
10            Cout : out STD_LOGIC);
11 end RippleCarry;
12
13 architecture Behavioral of RippleCarry is
14
15 signal IC:STD_LOGIC_VECTOR(nb downto 0);
16 signal p, g: STD_LOGIC_VECTOR(nb-1 downto 0);
17 begin
18     IC(0) <= Cin;
19     Cout <= IC(nb);
20     p<= A xor B;
21     MyFor: for i in 0 to nb-1 generate
22         S(i) <= p(i) xor IC(i);
23         IC(i+1) <= A(i) when (p(i)='0') else
24                         IC(i) when(p(i)='1') else 'X';
25     end generate MyFor;
26
27 end Behavioral;

```



# Codice VHDL

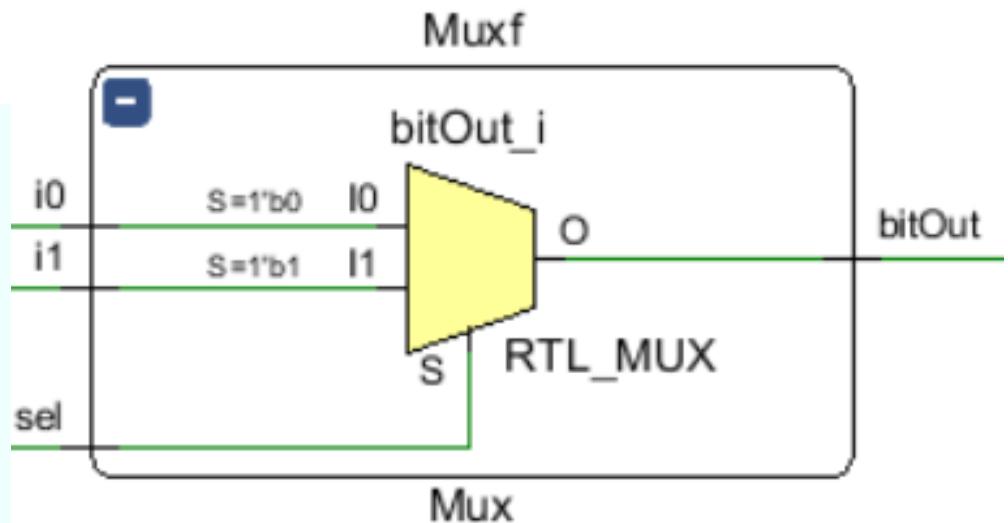
## Multiplexer 2:1

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Mux is
5     Port(i0: in STD_LOGIC;
6           i1: in STD_LOGIC;
7           sel: in STD_LOGIC;
8           bitOut:out STD_LOGIC);
9 end Mux;
10
11 architecture Behavioral of Mux is
12 begin
13
14     process(i0, i1, sel)
15     begin
16         case sel is
17             when '0' => bitOut<=i0;
18             when '1' => bitOut<=i1;
19             when others => bitOut<= 'X';
20         end case;
21     end process;
22
23 end Behavioral;
24

```

Schematic Design of a 2:1 Multiplexer (Mux di Riporti)



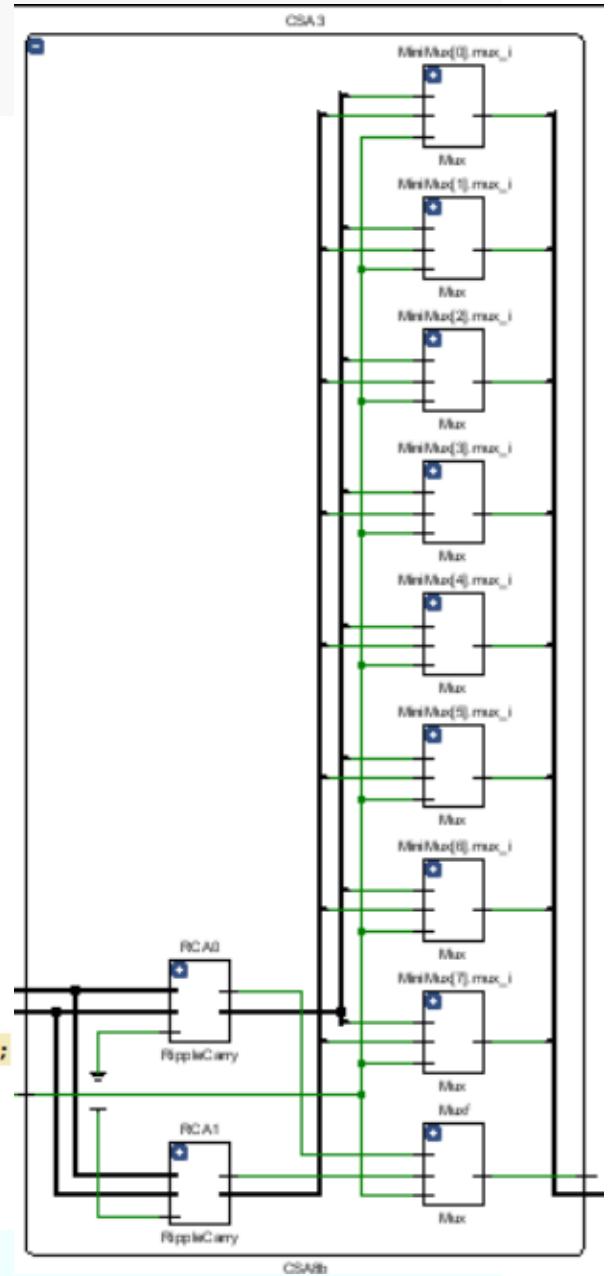
# Codice VHDL

## Carry Select a 8 bit

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity CSA8b is
5     Port ( Acs : in STD_LOGIC_VECTOR (7 downto 0);
6             Bcs : in STD_LOGIC_VECTOR (7 downto 0);
7             Carry_in : in STD_LOGIC;
8             Scs : out STD_LOGIC_VECTOR (7 downto 0);
9             Carry_out : out STD_LOGIC);
10 end CSA8b;
...
11
12 architecture Behavioral of CSA8b is
13
14 component RippleCarry is
15     generic(nb:integer:=8);
16     Port ( A : in STD_LOGIC_VECTOR (nb-1 downto 0);
17             B : in STD_LOGIC_VECTOR (nb-1 downto 0);
18             Cin : in STD_LOGIC;
19             S : out STD_LOGIC_VECTOR (nb-1 downto 0);
20             Cout : out STD_LOGIC);
21 end component;
22
23 component Mux is
24     Port(i0: in STD_LOGIC;
25           i1: in STD_LOGIC;
26           sel: in STD_LOGIC;
27           bitOut:out STD_LOGIC);
28 end component;
...
29
30 signal suml, sum0 : STD_LOGIC_VECTOR(7 downto 0);
31 signal carryl, carry0 : STD_LOGIC;
32 begin
33     --definisco i blocchi di RCA a 8 bit
34     RCA0: RippleCarry port map (Acs, Bcs, '0', sum0, carry0);
35     RCA1: RippleCarry port map (Acs, Bcs, '1', suml, carryl);
36
37     MiniMux: for i in 7 downto 0 generate
38         mux_i: Mux port map ( sum0(i), suml(i), Carry_in, Scs(i) );
39     end generate MiniMux;
40
41     Muxf: Mux port map (carry0, carryl, Carry_in, Carry_out);
42
43 end Behavioral;

```



Schematic Design of an 8 bit Carry Select Adder  
(CSA che rappresenta il 3° blocco)

# Codice VHDL

Carry Select a 32 bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CarrySelect32b is
    Port ( A_cs : in STD_LOGIC_VECTOR (31 downto 0);
           B_cs : in STD_LOGIC_VECTOR (31 downto 0);
           Cin_cs: in STD_LOGIC;
           S_cs : out STD_LOGIC_VECTOR (31 downto 0));
end CarrySelect32b;

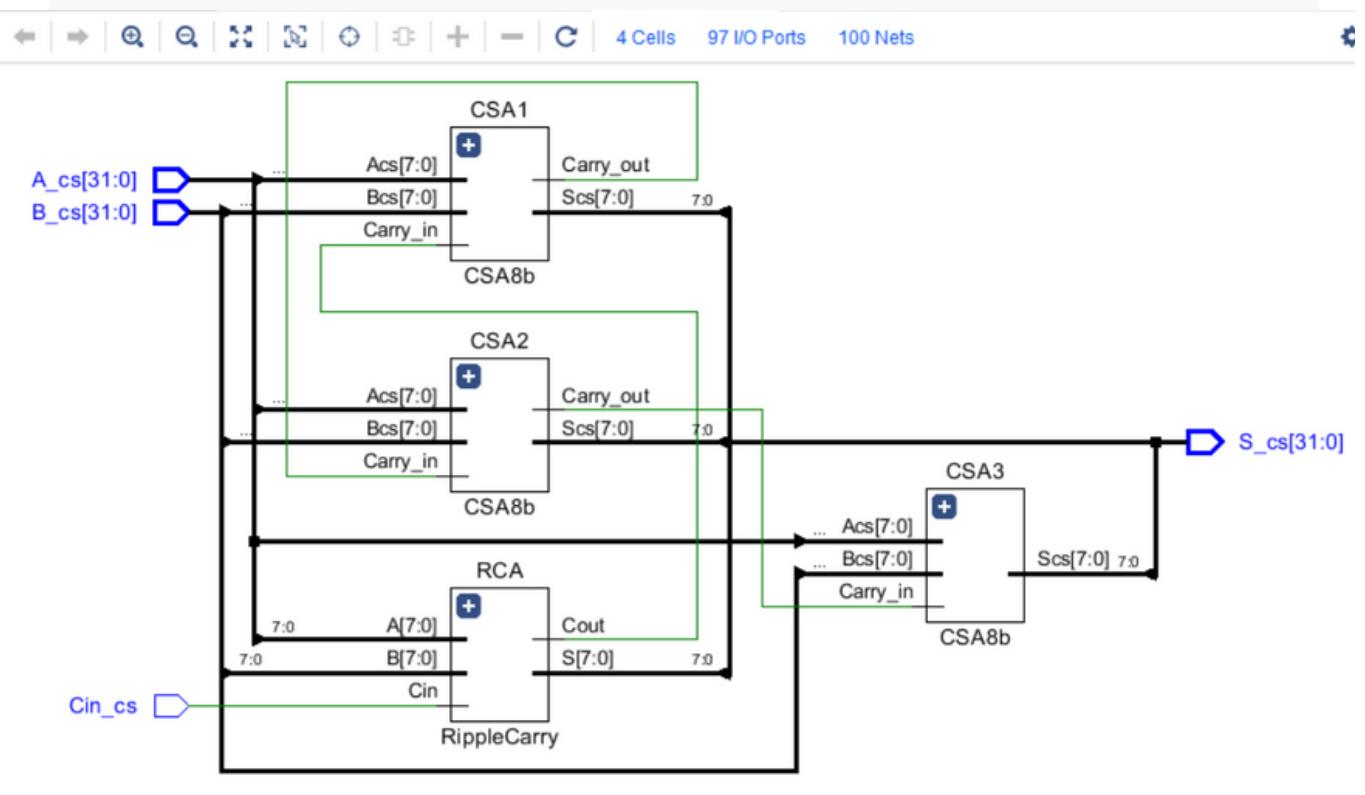
architecture Behavioral of CarrySelect32b is
component RippleCarry is
    generic(nb:integer:=8);
    Port ( A : in STD_LOGIC_VECTOR (nb-1 downto 0);
           B : in STD_LOGIC_VECTOR (nb-1 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (nb-1 downto 0);
           Cout : out STD_LOGIC);
end component;
component CSA8b is
    Port ( Acs : in STD_LOGIC_VECTOR (7 downto 0);
           Bcs : in STD_LOGIC_VECTOR (7 downto 0);
           Carry_in : in STD_LOGIC;
           Scs : out STD_LOGIC_VECTOR (7 downto 0);
           Carry_out : out STD_LOGIC);
end component;
signal Cout0, Cout1, Cout2, Cout3 : STD_LOGIC;
begin
    --primo blocco Ripple Carry per 8 bit meno significativi
    RCA: RippleCarry port map ( A_cs(7 downto 0), B_cs(7 downto 0), Cin_cs , S_cs(7 downto 0), Cout0 );

    -- blocchi di CSA ad 8 bit : 8x3=24 bit
    CSA1: CSA8b port map (A_cs(15 downto 8), B_cs(15 downto 8), Cout0, S_cs(15 downto 8), Cout1);
    CSA2: CSA8b port map (A_cs(23 downto 16), B_cs(23 downto 16), Cout1, S_cs(23 downto 16), Cout2);
    CSA3: CSA8b port map (A_cs(31 downto 24), B_cs(31 downto 24), Cout2, S_cs(31 downto 24), Cout3);

end Behavioral;

```

# Elaborated Design di CSA a 32 bit



# Test Bench 1

## Primo Test:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.all;
4
5 entity Sim_CSA32b is
6 end Sim_CSA32b;
7
8 architecture Behavioral of Sim_CSA32b is
9 component CarrySelect32b is
10    Port ( A_cs : in STD_LOGIC_VECTOR (31 downto 0);
11           B_cs : in STD_LOGIC_VECTOR (31 downto 0);
12           Cin_cs : in STD_LOGIC;
13           S_cs : out STD_LOGIC_VECTOR (31 downto 0));
14 end component;
15 signal IA, IB, Osum : STD_LOGIC_VECTOR(31 downto 0);
16 signal Icin : STD_LOGIC;
17
18 begin
19    --instantiate the Unit Under Test (UUT)
20    uut: CarrySelect32b port map(IA, IB, Icin, Osum);
21
22    Icin<= '0'; --Carry - in
23    --caso peggiore
24    IA <= "01010101010101010101010101010101"; --1431655765
25    IB <= "001010101010101010101010101010101"; --715827883
26
27 end Behavioral;

```

Un **test bench** è un file VHDL che implementa una simulazione di un circuito definendo dei valori dei segnali in ingresso. Esso, dunque, non viene sintetizzato ma risulta essere utile per valutare la correttezza del codice.

In questa prima simulazione, ho effettuato la somma di seguenti numeri binari a 32 bit:

- = "01010101010101010101010101010101" pari a 1.431.655.765 nella rappresentazione unsigned, e
- = "001010101010101010101010101010101" pari a 715.827.883

Behavioral Simulation of the first test:



INFO: [USF-XSim-97] XSim simulation ran for 1000ns  
launch\_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:09 . Memory (MB): peak = 2170.027 ; gain = 0.000

# Test Bench 2

## Secondo Test

```

entity Sim_CSA32b is
end Sim_CSA32b;

architecture Behavioral of Sim_CSA32b is
component CarrySelect32b is
    Port ( A_cs : in STD_LOGIC_VECTOR (31 downto 0);
           B_cs : in STD_LOGIC_VECTOR (31 downto 0);
           Cin_cs: in STD_LOGIC;
           S_cs : out STD_LOGIC_VECTOR (31 downto 0));
end component;
signal IA, IB, Osum : STD_LOGIC_VECTOR(31 downto 0);
signal Icin : STD_LOGIC;

begin
    --instantiate the Unit Under Test (UUT)
    uut: CarrySelect32b port map(IA, IB, Icin, Osum);
    --simulazione esaustiva
    ICin <= '0';
    CSA: PROCESS BEGIN
        for i in 1073741820 to 1073741830 loop
            IA<= CONV_STD_LOGIC_VECTOR(i, 32);
            for j in 2147483630 to 2147483640 loop
                IB<= CONV_STD_LOGIC_VECTOR(j, 32);
                wait for 20ns;
            end loop;
        end loop;
    END PROCESS;

end Behavioral;

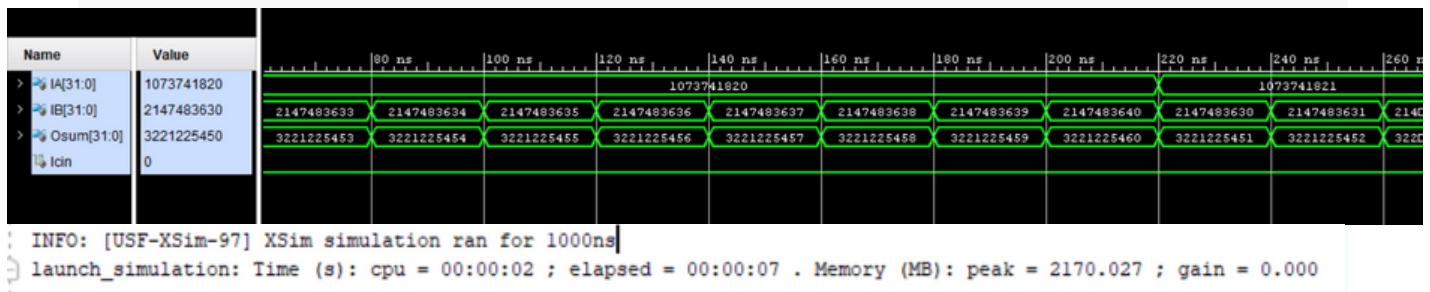
```

In questa seconda simulazione, ho effettuato un test inviando in input tutte le possibili combinazioni di operandi tra il range di valori che va da [0 to 4294967295] in quanto stiamo considerando i numeri unsigned.

Per il primo input, ho scelto i valori che vanno da 1073741820 a 1073741830 mentre per il secondo input, i valori vanno da 2147483630 a 2147483640

# Simulation

## Behavioral Simulation



Facendo partire la simulazione, osserviamo che non c'è nessun errore di calcolo dovuto al fatto che la Behavioral simulation non tiene in considerazione delle sprecifiche reali di un circuito.

# Simulation

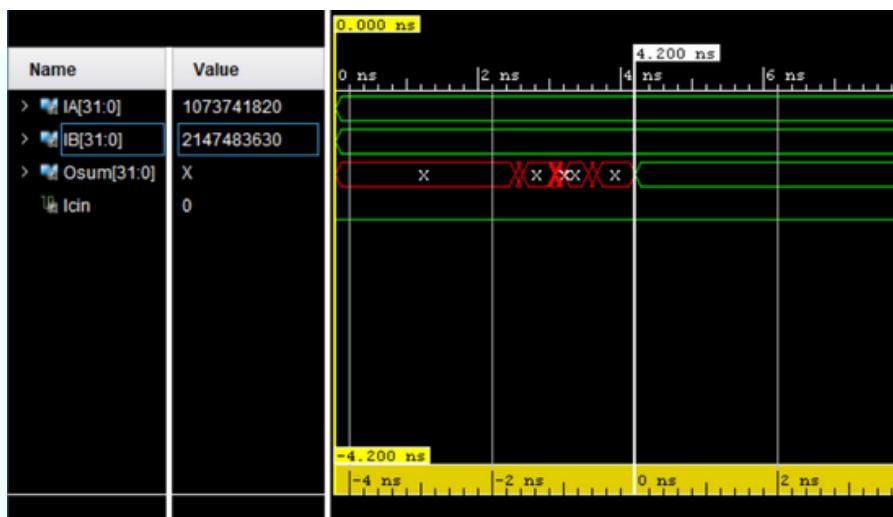
## Post- Synthesis Timing Simulation



Dopo aver effettuato il Synthesis, la logica generale della progettazione viene sintetizzato nelle specifiche primitive del dispositivo rendendo disponibile il routing stimato e i ritardi dei componenti.

Attraverso la Post- Synthesis Timing Simulation, notiamo la presenza di una specificazione 'X' la quale indica che vi è una elaborazione degli ingressi in corso, evidenziando dunque che un ritardo dovuto ai componenti.

In questo test, il ritardo iniziale sarà pari a 4,200ns dovuto al fatto che PST simulation utilizza il timing delay stimato dalla macchina ma non tiene in considerazione il ritardo di interconnessione.



# Caratterizzazione del progetto in termini di risorse utilizzate

## 1. Numeri di LUT utilizzati

Operando la Post-Implementation Simulation è possibile visualizzare quali componenti del device a nostra disposizione sono utilizzati dal circuito. In particolare, il nostro device fa uso di Look Up Table, detto anche LUT che sono delle strutture dati utilizzati per memorizzare sia le tabelle logiche delle funzioni logiche, sia per memorizzare i dati.

Grazie al meccanismo delle LUT, non è più necessario operare i calcoli molto complessi andando da capo, bensì, basta accedere ad un area di memoria..

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	53	0	53200	0.10
LUT as Logic	53	0	53200	0.10
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

Di queste LUT, 35 sono state utilizzate per memorizzare una sola tabella di verità (1 uscita) e 18 invece sono state impiegate per memorizzare funzioni logiche a 2 uscite.

Questo report ci dice che sono state utilizzate 53 LUT, tutte utilizzate per svolgere operazioni logiche, ovvero per memorizzare tabelle di verità.

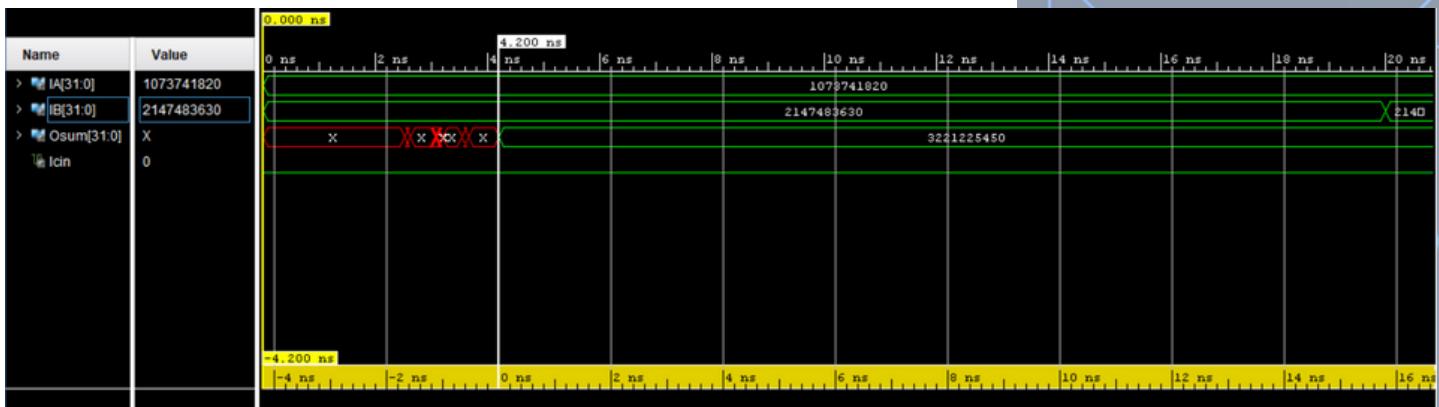
2. Slice Logic Distribution

Site Type	Used	Fixed	Available	Util%
Slice	20	0	13300	0.15
SLICEL	5	0		
SLICEM	15	0		
LUT as Logic	53	0	53200	0.10
using O5 output only	0			
using O6 output only	35			
using O5 and O6	18			
LUT as Memory	0	0	17400	0.00
LUT as Distributed RAM	0	0		
LUT as Shift Register	0	0		
LUT Flip Flop Pairs	0	0	53200	0.00
Unique Control Sets	0	0		

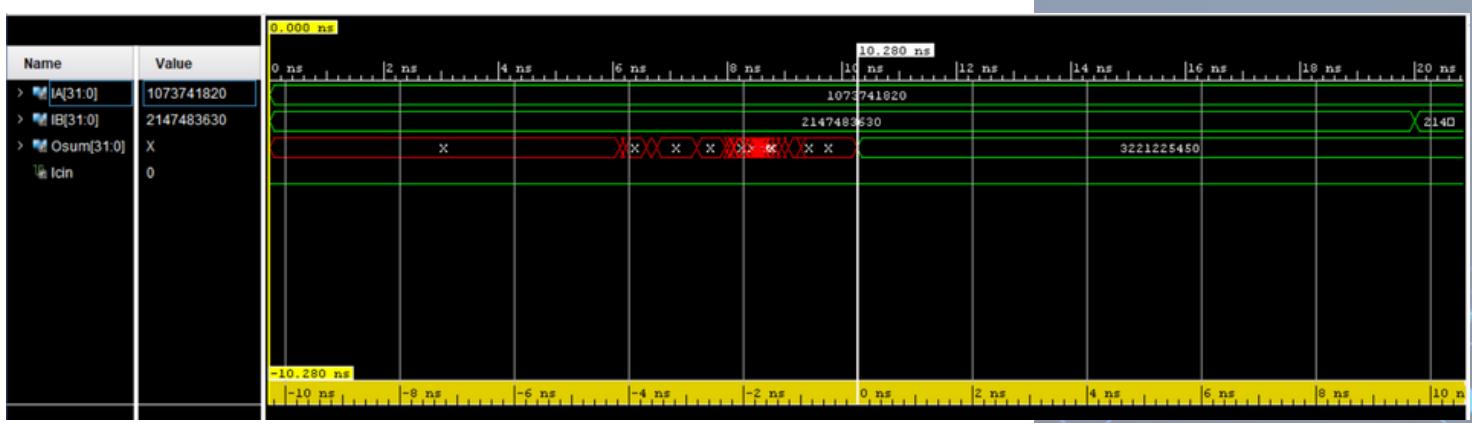
\* Note: Review the Control Sets Report for more information regarding control sets.

## 2. Tempo Impiegato dal circuito a svolgere le operazioni descritte in VHDL

# Post-Synthesis Timing Simulation vs. Post-Implementation Timing Simulation



Come detto precedentemente, facendo il Post-Synthesis Timing Simulation, viene considerato il ritardo duto ai componenti. In questa simulazione, osserviamo un ritardo iniziale pari a 4,2 ns



Mentre con il Post-Implementation Timing Simulation, osserviamo un ritardo iniziale pari a 10,28ns, ovvero maggiore di quello ottenuto tramite la PST simulation dovuto al fatto che in questa fase di simulazione, viene aggiunto il ritardo a causa di routing