Corso di Elettronica Digitale. Risafi Natale Francesco Pio, Matricola 223491

Relazione del progetto: Carry-Select Adder a 32 bit

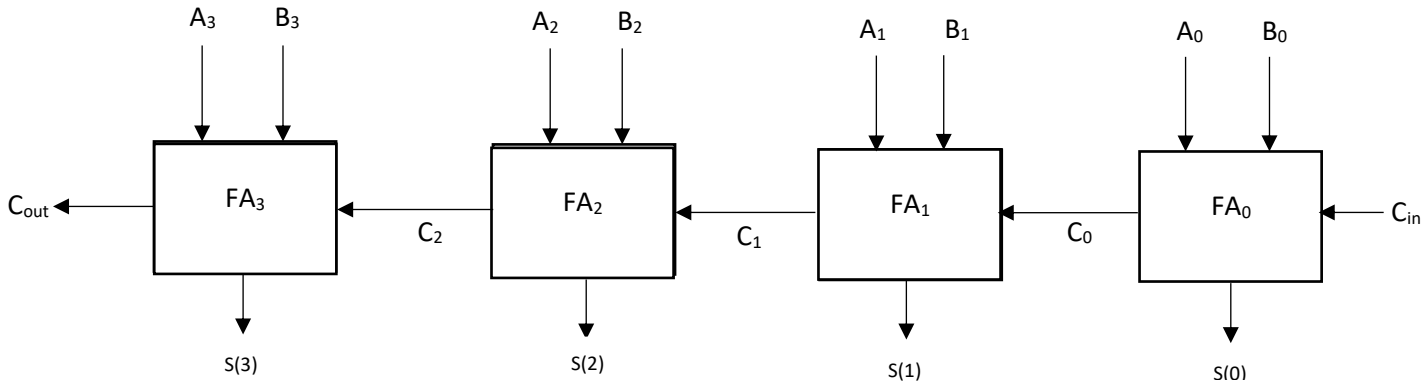
Prima di implementare ed effettuare le varie simulazioni relative a questo circuito, inizio con la sua descrizione; un *Carry-Select Adder* è uno dei circuiti sommatori più veloci, usato in moltissimi processori di data-processing, per realizzare molte funzioni aritmetiche. Inoltre, per descrivere questo tipo di circuito, avremo bisogno di un altro tipo di circuito sommatore, questa volta però" componente", in quanto lo useremo per realizzare, in maniera ridondante, il nostro Carry-Select Adder. Il circuito in questione è il Ripple-Carry Adder; il nostro circuito sarà sicuramente più veloce di quest'ultimo, ma come già detto ci servirà per descrivere quello che sarà il nostro Carry-Select Adder a 32 bit.

Infatti, sarà composto da:

- 1 RCA (Ripple-Carry Adder a 4 bit);
- 7 CSLA (Carry-Select Adder a 4 bit).

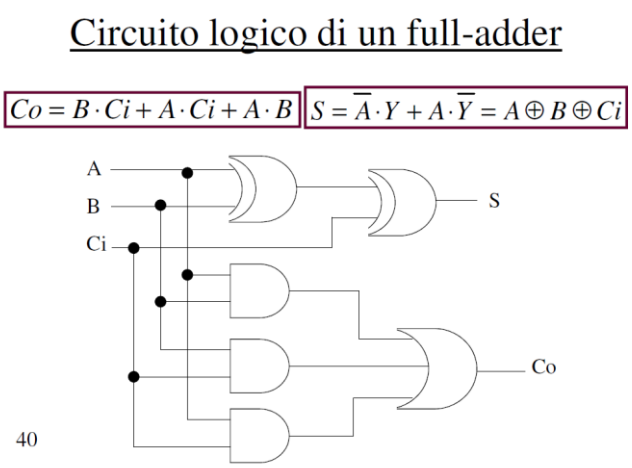
Inoltre, ciascuno dei nostri Carry-Select, che andranno a comporre il circuito-sommatore a 32 bit, conterrà al suo interno un RCA. Il RCA è il sommatore più semplice e meno costoso, in quanto richiede meno porte logiche di ciascun altro tipo di circuito. Dunque, analizzando componente per componente il nostro sommatore a 32 bit, alcuni dei componenti di base saranno sicuramente gli RCA, ciascuno dei quali, essendo a 4 bit, saranno composti da 4 Full-Adder.

Schema di ognuno dei nostri Ripple-Carry Adder:



Come mostrato sopra ognuno dei nostri Full-Adder, che compongono il nostro generico Ripple-Carry Adder, sarà a 2 bit e sarà composto da 3 porte AND, 1 porta OR e 2 porte XOR. Dunque, stiamo cercando di scomporre il nostro circuito in "sotto-circuiti" più semplici in maniera tale da vedere come agisce dietro le quinte il nostro sommatore a 32 bit, e dare una descrizione più specifica di ogni singola componente, che verrà poi simulata con VIVADO. Per descrivere i vari circuiti con VIVADO userò il linguaggio VHDL (***V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage)* che è un linguaggio di DESCRIZIONE, utile per definire al meglio le varie funzioni che sono svolte da diversi tipi di circuiti. Inoltre, ha la caratteristica di essere un linguaggio "concorrente", dove quindi possiamo usare una variabile ancora prima di dichiararla. Non sempre sarà un linguaggio concorrente, infatti ci saranno alcuni costrutti che lo renderanno "sequenziale", ad esempio *process*, dove sarà dunque importante l'ordine delle varie istruzioni e la dichiarazione delle varie variabili. Al posto del bit come tipo di Input/Output delle nostre simulazioni useremo lo **STANDARD LOGIC**, che sarà possibile usarlo importando la libreria *library IEEE*, e che renderà più flessibile la scrittura del codice, oltre ad ampliare la possibilità di rappresentazione. Inoltre, in VIVADO per descrivere il nostro circuito dovremo definire la **ENTITY** nella quale andremo a descrivere ingressi e uscite e dovremo definire anche la **ARCHITECTURE** nella quale useremo il codice VHDL per descrivere le varie funzioni del nostro circuito ed indicare come sono rappresentate le varie componenti.

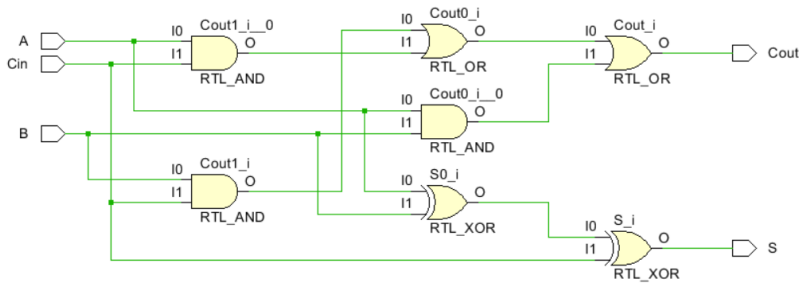
Iniziamo descrivendo proprio uno degli elementi di base del nostro circuito, il FULL-ADDER:



Il Full-Adder sarà caratterizzato da 3 ingressi e 2 uscite. Esso potrà sommare 3 bit, i due bit definiti da A e B ed in più il bit di riporto. Restituendo alla fine un bit di Somma (S), più un bit di riporto in uscita (C_{out}). Vediamo una sua implementazione in VIVADO:

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

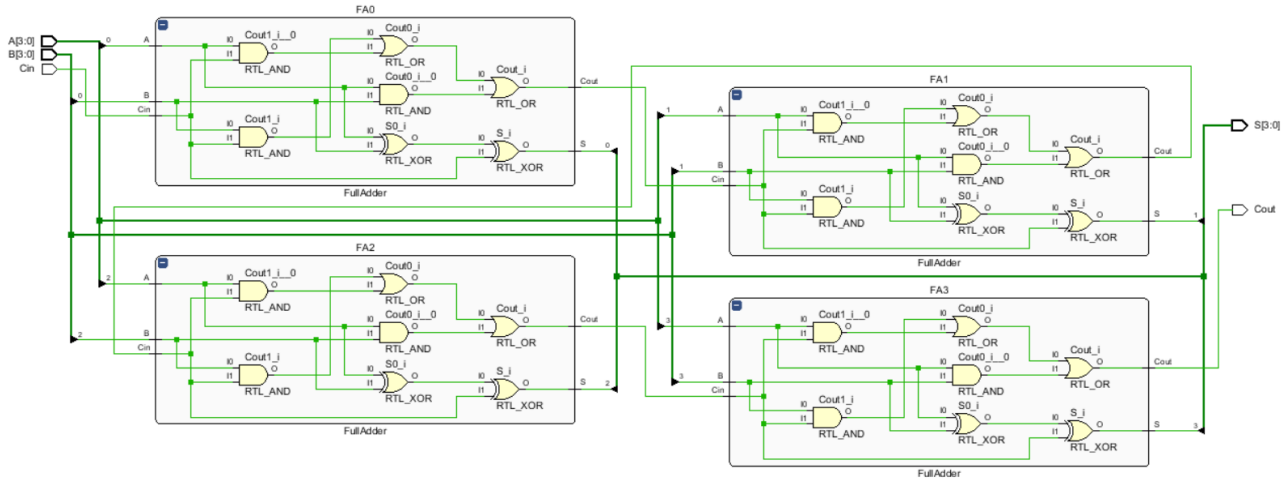
E volendo verificare che quello che abbiamo scritto in VHDL, definisce il nostro circuito, che ci aspettiamo, possiamo verificarlo attraverso una RTL analysis:



Continuando a definire gli elementi che compongono il nostro circuito sommatore a 32 bit, importante è il compito eseguito dai singoli Ripple-Carry, tutti a 4 bit, inoltre questo sommatore non fa altro che calcolare la somma con il metodo *"carta e penna"*; il nostro RCA avrà dunque due ingressi a 4 bit, più un bit di ingresso di riporto (Cin), per poi produrre 2 uscite, una a 4 bit che sarà il vettore somma S risultante e l'altro il bit di riporto (Cout); passando alla sua implementazione in VIVADO:

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

entity RippleCarryAdder is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (3 downto 0);
           Cout : out STD_LOGIC);
end RippleCarryAdder;

architecture Behavioral of RippleCarryAdder is
signal c: STD_LOGIC_VECTOR(2 downto 0);

component FullAdder
    port(A: in STD_LOGIC;
         B: in STD_LOGIC;
         Cin: in STD_LOGIC;
         Cout: out STD_LOGIC;
         S: out STD_LOGIC);
end component;

begin
    FA0: FullAdder port map(A(0), B(0), Cin, c(0), S(0));
    FA1: FullAdder port map(A(1), B(1), c(0), c(1), S(1));
    FA2: FullAdder port map(A(2), B(2), c(1), c(2), S(2));
    FA3: FullAdder port map(A(3), B(3), c(2), Cout, S(3));

end Behavioral;
```

Abbiamo dunque la necessità di usare altre parole chiavi, come *signal*, *component* e *port map;* ognuna di queste usate nella nostra Architecture. Usiamo *signal* per definire un nuovo vettore di STD_LOGIC, che serve proprio per tenere traccia dei vari segnali di riporto intermedi che si trovano tra i vari FullAdder, e quindi essi non hanno una direzione, ma solo il compito di collegare una uscita con un ingresso. Una versione alternativa per l'implementazione sarebbe stata possibile usando quello che è il costrutto *for…in…to…generate*, avendo già definito questo vettore C che con i suoi indici i, avrebbe tenuto conto di ogni ingresso e di ogni uscita; in questo caso però essendo che ogni RCA è a 4 bit non è necessario, ed è anche poco dispendioso eseguirlo per normale assegnazione, come nel codice sopra riportato. Usiamo invece *component* per "richiamare" il nostro circuito FullAdder precedentemente definito, riportando esattamente tutte le porte definite in precedenza. *port map* viene invece usato per la costruzione dei vari FullAdder, assegnando ad ognuno di essi i vari bit STD_LOGIC che dovranno ricevere come ingresso e come uscita. Eseguendo una RTL analysis e generando un suo Schematic:



Notiamo come ogni bit C(i), in uscita dai vari FullAdder, vanno in ingresso come riporto ai vari FullAdder successivi.

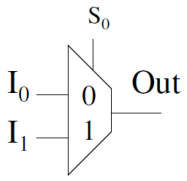
Dunque, tornando al nostro circuito di partenza, il Carry-Select Adder a 32 bit, per implementarlo abbiamo deciso di partire dalle componenti di base che lo compongono per poi rappresentarlo in VIVADO attraverso

le sue varie simulazioni. Inoltre, ricordando che l'abbiamo definito come composto da un RCA a 4 bit e 7 CSLA a 4 bit, dobbiamo ora implementare in VIVADO un singolo Carry-Select Adder a 4 bit.

Come già detto il Carry-Select Adder è un circuito sommatore sicuramente più veloce del Ripple-Carry Adder, ma anche più costoso; infatti, in esso è presente un certo "parallelismo" che fa in modo di velocizzare i calcoli ma è necessaria l'aggiunta di multiplexer, che avranno il compito di selezionare le uscite di questi blocchi, e riceveranno in ingresso come segnale di selezione il bit di riporto del blocco precedente. In particolare, i nostri Carry-Select Adder a 4 bit saranno composti da due Ripple-Carry Adder in parallelo e 5 Multiplexer 2_1, 4 dei quali gestiranno il bit risultante ed uno gestirà l'ultimo bit di riporto. I multiplexer 2_1 saranno formati da 2 porte AND 1 porta OR ed 1 porta XOR. Inoltre, vale che, in un Carry-Select Adder ad n-bit realizzato usando blocchi Ripple-Carry a 4 bit, il numero di stadi di Multiplexer necessari alla realizzazione del circuito è $\frac{n}{4} - 1$.

Continuando ad usare la logica applicata finora per descrivere il circuito, prima di implementare il Carry-Select finale, passo alla implementazione del MUX 2_1. Il bit di selezione S_0 determina quale dei due bit passati come ingresso deve essere uscita del nostro MUX.

Multiplexer 2:1



$$Out = I_0 \cdot \overline{S_0} + I_1 \cdot S_0$$

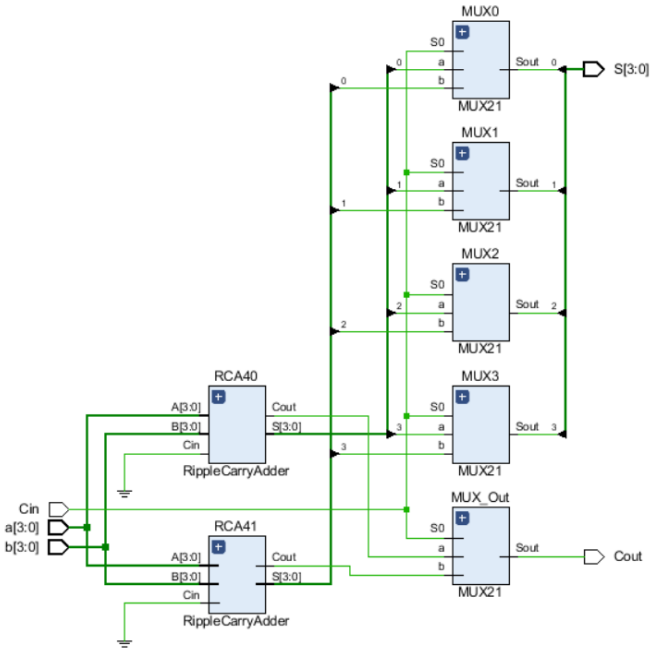
Dunque, definisco la sua corrispondente implementazione in VIVADO:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity MUX21 is
5      Port ( a : in STD_LOGIC;
6             b : in STD_LOGIC;
7             S0 : in STD_LOGIC;
8             Sout : out STD_LOGIC);
9  end MUX21;
10
11 architecture Behavioral of MUX21 is
12
13 begin
14     Sout <= a when (S0 = '0') else
15             b when (S0 = '1') else 'X';
16
17 end Behavioral;
```

Posso adesso passare a definire ed implementare l'ultimo elemento che ci consentirà di comporre il nostro sommatore. Implemento in VIVADO un Carry-Select Adder a 4 bit, che insieme ad altri come descritto sopra andrà a comporre il nostro sommatore finale. Per implementare questo circuito in VIVADO useremo dunque 4 MUX2_1 che gestiranno le somme dei vari Ripple-Carry, 1 MUX2_1 che gestirà i Carry dei due RCA a 4 bit. I due RCA si occuperanno di fare la somma "parallelamente".

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

entity CSLA4bit is
    Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (3 downto 0);
           Cout : out STD_LOGIC);
end CSLA4bit;

architecture Behavioral of CSLA4bit is
signal p,q: STD_LOGIC_VECTOR (3 downto 0);
signal c0,c1: STD_LOGIC; --Carry Out in uscita dai due RCA.

component MUX21
        port( a : in STD_LOGIC;
              b : in STD_LOGIC;
              S0 : in STD_LOGIC;
              Sout : out STD_LOGIC);
end component;

component RippleCarryAdder
        port(  A : in STD_LOGIC_VECTOR (3 downto 0);
               B : in STD_LOGIC_VECTOR (3 downto 0);
               Cin : in STD_LOGIC;
               S : out STD_LOGIC_VECTOR (3 downto 0);
               Cout : out STD_LOGIC);
end component;
begin
    --RCA a 4 bit
    RCA40: RippleCarryAdder
        port map(a(3 downto 0), b(3 downto 0), '0', p, c0);
    RCA41: RippleCarryAdder
        port map(a(3 downto 0), b(3 downto 0), '1', q, c1);

    --MUX2_1
    MUX0: MUX21
        port map(p(0), q(0), Cin, S(0));
    MUX1: MUX21
        port map(p(1), q(1), Cin, S(1));
    MUX2: MUX21
        port map(p(2), q(2), Cin, S(2));
    MUX3: MUX21
        port map(p(3), q(3), Cin, S(3));
    MUX_Out: MUX21
        port map(c0, c1, Cin, Cout);
end Behavioral;
```

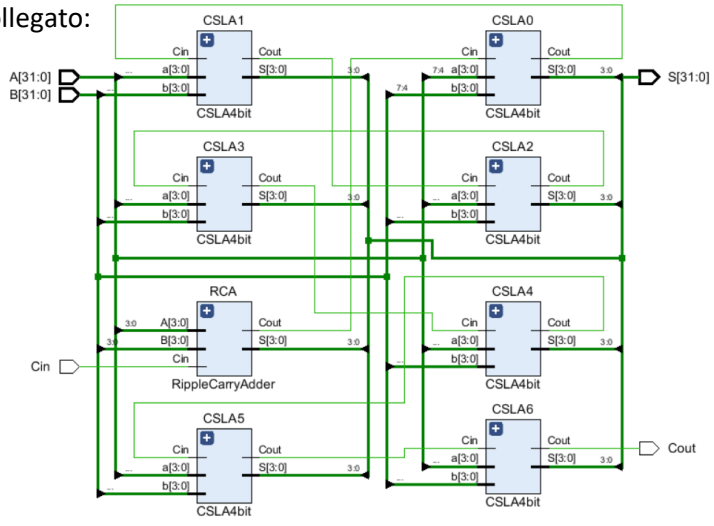
Una volta descritta in VIVADO la funzione logica del nostro Carry-Select Adder a 4 bit, rappresentiamo il suo schematic:



Una volta implementato il Carry-Select Adder a 4 bit, possiamo comporre il nostro sommatore Carry-Select Adder a 32 bit, come già detto in precedenza, composto da 7 Carry-Select Adder a 4 bit in serie con 1 Ripple-Carry Adder a 4 bit. Per prima cosa scrivo la sua implementazione in VIVADO:

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

entity CSLA32bit is
    Port ( A : in STD_LOGIC_VECTOR (31 downto 0);
           B : in STD_LOGIC_VECTOR (31 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (31 downto 0);
           Cout : out STD_LOGIC);
end CSLA32bit;

architecture Behavioral of CSLA32bit is
signal p: STD_LOGIC_VECTOR (7 downto 0); --per gestire i riporti tra i vari componenti
signal s1: STD_LOGIC_VECTOR(31 downto 0); --segnale delle somme parziali

component CSLA4bit
     port (a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (3 downto 0);
           Cout : out STD_LOGIC);
end component;

component RippleCarryAdder
     port (A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (3 downto 0);
           Cout : out STD_LOGIC);
end component;

begin
    --RCA
    RCA: RippleCarryAdder
    port map(A(3 downto 0), B(3 downto 0), Cin, s1(3 downto 0), p(0));

    --Implementazione dei 7 CSLA che comporranno il sommatore
    CSLA0: CSLA4bit
        port map(A(7 downto 4), B(7 downto 4), p(0), s1(7 downto 4), p(1));
    CSLA1: CSLA4bit
        port map(A(11 downto 8), B(11 downto 8), p(1), s1(11 downto 8), p(2));
    CSLA2: CSLA4bit
        port map(A(15 downto 12), B(15 downto 12), p(2), s1(15 downto 12), p(3));
    CSLA3: CSLA4bit
        port map(A(19 downto 16), B(19 downto 16), p(3), s1(19 downto 16), p(4));
    CSLA4: CSLA4bit
        port map(A(23 downto 20), B(23 downto 20), p(4), s1(23 downto 20), p(5));
    CSLA5: CSLA4bit
        port map(A(27 downto 24), B(27 downto 24), p(5), s1(27 downto 24), p(6));
    CSLA6: CSLA4bit
        port map(A(31 downto 28), B(31 downto 28), p(6), s1(31 downto 28), p(7));
    Cout <= p(7);
    S <= s1;
end Behavioral;
```

Dunque, vediamo il nostro sommatore come una scomposizione di circuiti più semplici dove come *signal* ho introdotto 2 vettori: p di 8 elementi, che serve per gestire i vari riporti tra i vari Carry-Select a 4 bit che compongono il circuito, e s1 che ne gestisce le varie somme parziali, infatti, alla fine posso pensare di mandare come Output di somma proprio il vettore s1, che assegnerò al vettore S, in quanto sarò sicuro che saranno entrambi vettori di 32 elementi. Possiamo verificare attraverso la costruzione di uno Schematic, come il nostro circuito è collegato:

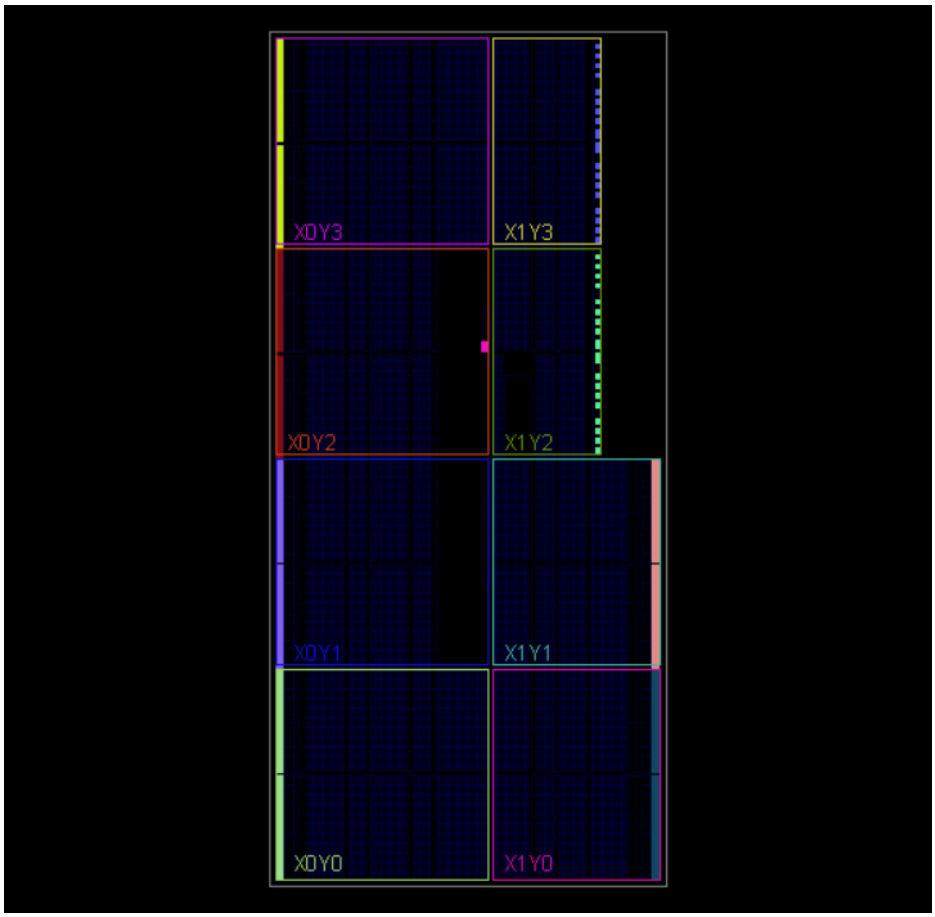


Una volta costruito il nostro circuito, per testarne funzionamento ed ulteriori performance che può garantire attraverso test, dovremo definire un nuovo file, questa volta di simulazione, cioè il TEST-BENCH, la cui una delle regole principale è che ha la entity vuota:

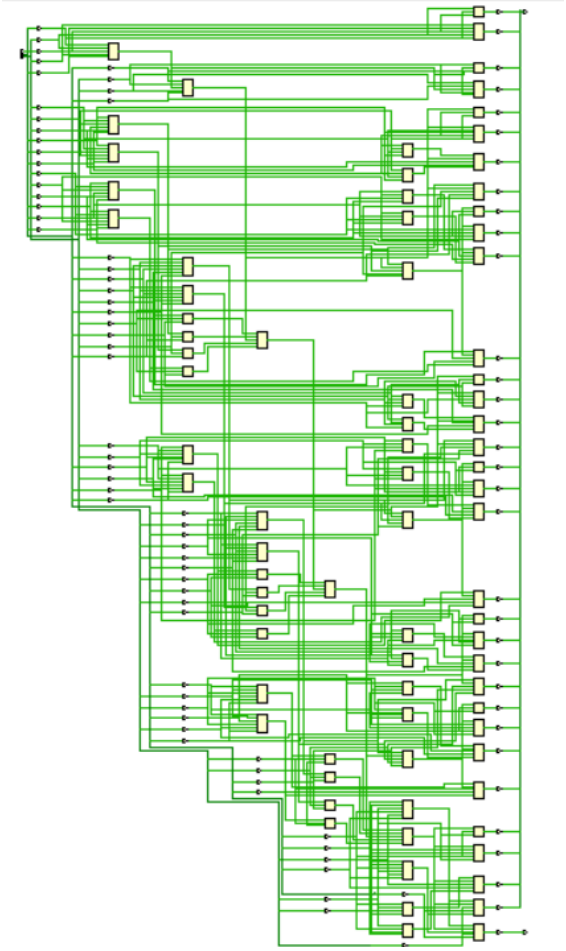
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL; --per effettuare le conversioni

entity SimCSLA32bit is
-- Una delle regole di un Test-Bench è che la entity deve essere vuota
end SimCSLA32bit;

architecture Behavioral of SimCSLA32bit is
component CSLA32bit is
    Port ( A : in STD_LOGIC_VECTOR (31 downto 0);
           B : in STD_LOGIC_VECTOR (31 downto 0);
           Cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (31 downto 0);
           Cout : out STD_LOGIC);
end component;
--Inoltre nel Test-Bench serve un segnale per ogni porta
signal iA : STD_LOGIC_VECTOR (31 downto 0);
signal iB : STD_LOGIC_VECTOR (31 downto 0);
signal iCin : STD_LOGIC;
signal oS : STD_LOGIC_VECTOR (31 downto 0);
signal oCout : STD_LOGIC;

begin
    CIRC: CSLA32bit port map(iA, iB, iCin, oS, oCout);
-- definisco adesso i miei segnali, le forme d'onda
-- l'ordine nel process sarà importante perchè sarà sequenziale
      iCin <= '0';
      process begin
         for i in 50 to 150 loop
            iA <= conv std logic vector(i,32);
            for j in 150 downto 50 loop
               iB <= conv_std_logic_vector(j,32);
               wait for 10ns;
            end loop;
         end loop;
   end process;


end Behavioral;
```

Nella scrittura di questo Test-Bench troviamo l'uso di una nuova parola chiave di VIVADO, che è *process*, che indica l'inizio di un nuovo tipo di costrutto, che differentemente dalle generalità di VIVADO, è di tipo sequenziale; dunque, è importante l'ordine di come vengono definite le istruzioni in questa struttura. Troviamo altre parole chiave come wait for, che viene usato proprio in tipi di costrutto sequenziali e dice quanto tempo deve passare da una istruzione alla successiva. Fondamentale con quest'ultima istruzione è l'uso del for…in…loop, che permette di eseguire per n volte un gruppo di statement sequenziali. I segnali verranno rappresentati come delle forme d'onda, che andranno a definire il nostro output di Somma(oS) e quello di riporto(oCout). Adesso possiamo eseguire una Sintesi per vedere come verrà rappresentato il nostro circuito come Device e come attraverso le LUT (Look Up Table), che sono degli elementini di memoria che memorizzano una qualsiasi porta logica che presenta al massimo 6 ingressi ed una sola uscita, inoltre andranno a comporre un chip fpga che è proprio identificato come una matrice di LUT, e le rispettive tabelle di verità.

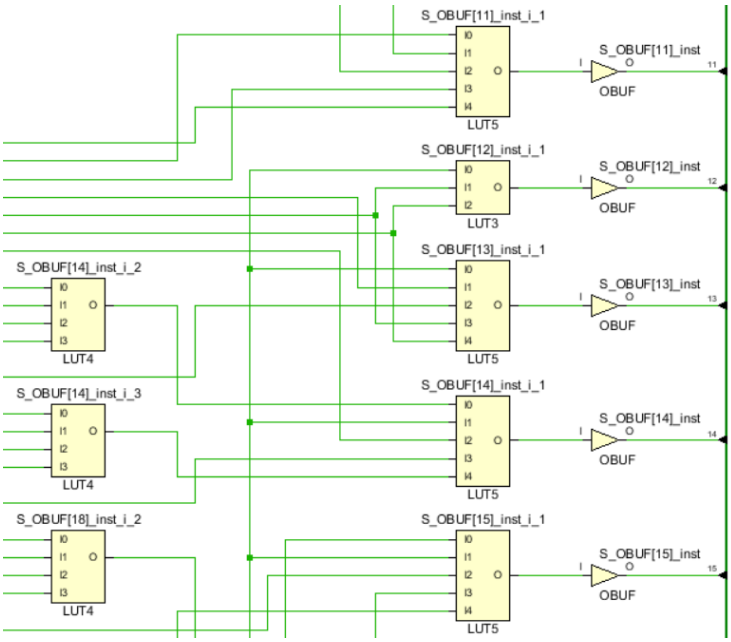
DEVICE(Post-Synthesis)



SCHEMATIC (usando le LUT)



Facendo uno zoom su una parte del circuito notiamo come ogni parte del nostro circuito è stato sostituito con i rispettivi buffer:



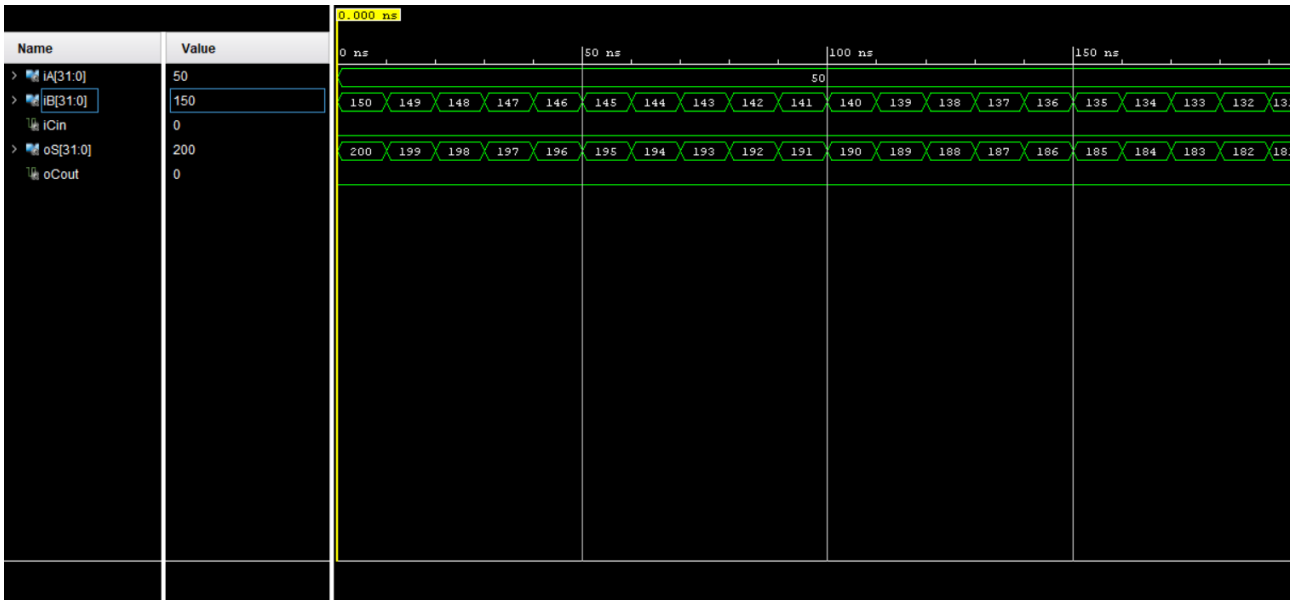
Selezionando una di queste LUT (ad esempio la S_OBUF[14]_inst_i_2/LUT4) possiamo vedere alcune delle caratteristiche di quella LUT, tra cui la funzione logica che descrive e la relativa tabella di verità:

S_OBUF[14]_inst_i_2

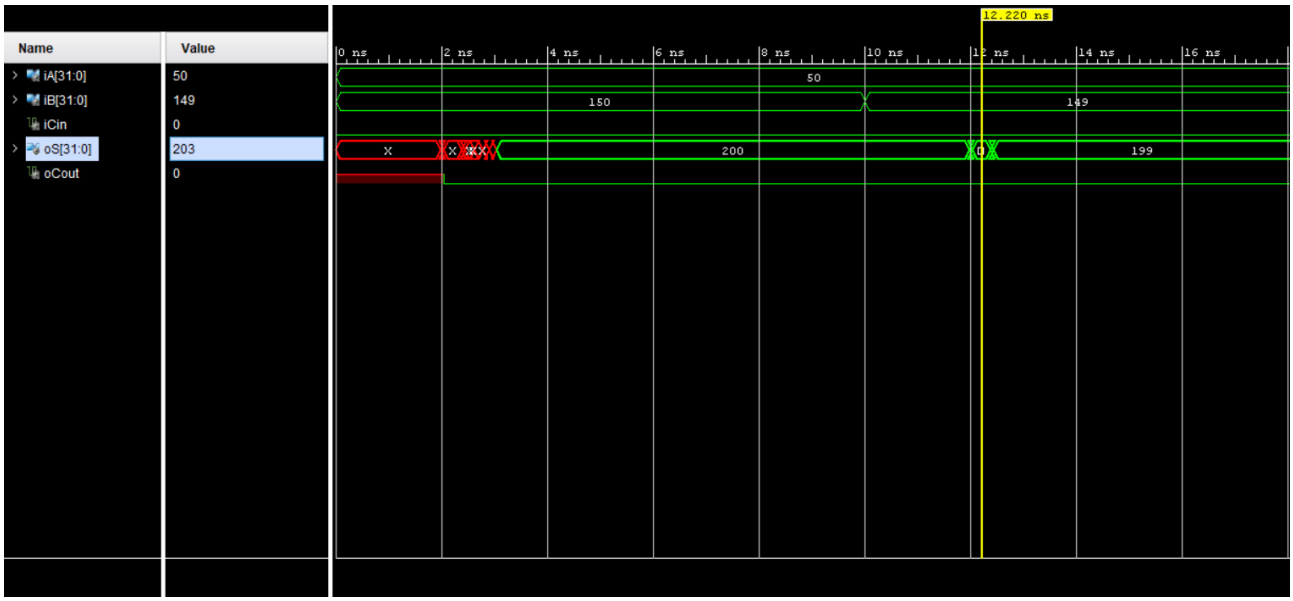
I3	I2	I1	I0	O=I0 & I2 + I1 & I2 + I0 & !I2 & I3 + I1 & !I2 & I3 + !I1 & I2 & I3
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

O=I0 & I2 + I1 & I2 + I0 & !I2 & I3 + I1 & !I2 & I3 + !I1 & I2 & I3

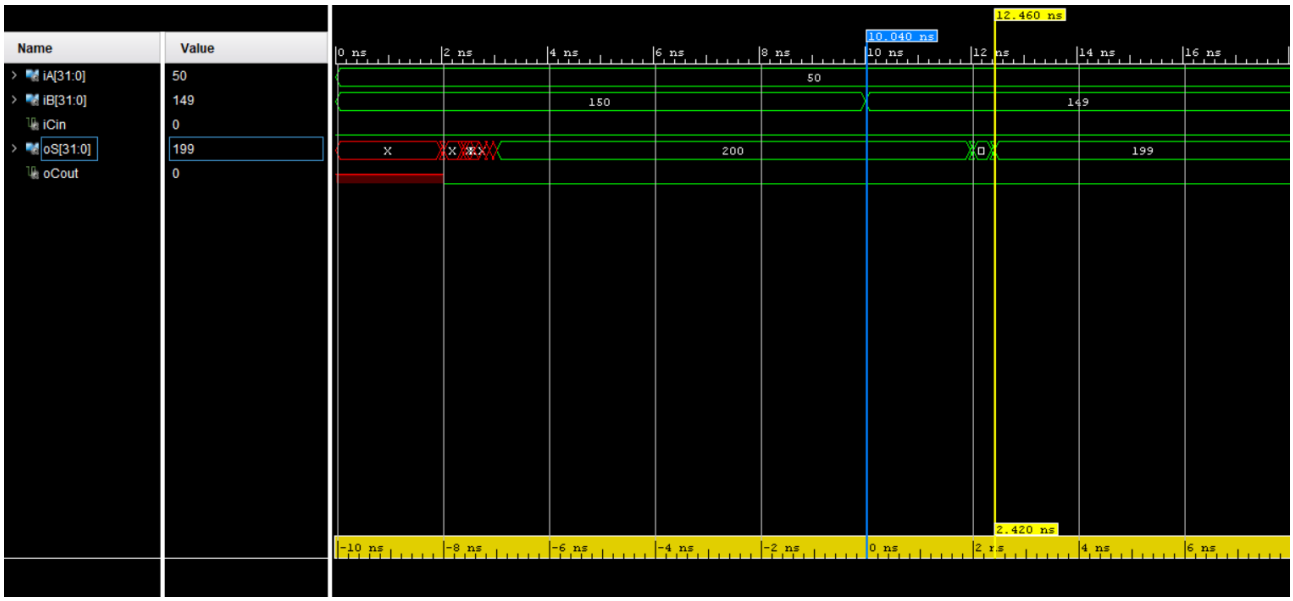
Una volta eseguita una sintesi possiamo verificare il corretto funzionamento del nostro sommatore attraverso delle simulazioni:



Verifichiamo dunque la correttezza di questi calcoli, che verranno applicati sequenzialmente, come descritto nel for precedentemente implementato; in questo caso decido di rappresentare tutto in base decimale, il primo addendo è quello dato dall'ingresso iA, il secondo è quello dato dall'ingresso iB, mentre oS sarà il nostro vettore di 32 bit risultante, dato dalla somma dei due. Una volta eseguita la sintesi del nostro circuito, possiamo effettuare una nuova simulazione, questa volta Post-Synthesis, di tipo Timing, in maniera tale da calcolare i ritardi in maniera reale:



Notiamo subito un segno rosso all'interno della nostra simulazione, nella fase iniziale, la somma è indeterminata, perché l'operazione non è ancora finita, sono arrivati degli ingressi ma prima di vedere l'uscita devo aspettare un certo tempo. Invece, nella zona dove è inserito il marker, a 12,220 ns, notiamo che il valore della somma 50+149 non è corretto, tutti i valori che assume la somma in quella porzione di tempo sono dei normali assestamenti dell'uscita, non è passato un tempo adeguatamente lungo per effettuare le propagazioni del riporto per il calcolo del risultato corretto, e questi prendono il nome di Glitch. Possiamo anche verificare questo ritardo necessario per il calcolo corretto di una operazione ad esempio nel nostro caso, per calcolare correttamente 50+149, da quando riceve il bit di 149, necessita di un tempo pari a 2,420 ns:



Per completare la caratterizzazione del circuito, una volta esaminato il tempo impiegato per eseguire alcune funzioni, dobbiamo studiare il numero di LUT ed in generale di risorse utilizzate; questa informazione si trova in un report specifico: impl_1_place_report_utilization_0. Verrà aperto un file di testo che conterrà in dettaglio tutte le informazioni fornite.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	55	0	41000	0.13
LUT as Logic	55	0	41000	0.13
LUT as Memory	0	0	13400	0.00
Slice Registers	0	0	82000	0.00
Register as Flip Flop	0	0	82000	0.00
Register as Latch	0	0	82000	0.00
F7 Muxes	0	0	20500	0.00
F8 Muxes	0	0	10250	0.00

Da qui vediamo che utilizziamo 55 LUT e tutte saranno usate come funzione logica, attraverso le tabelle di verità.

Site Type	Used	Fixed	Available	Util%
Slice	19	0	10250	0.19
SLICEL	19	0		
SLICEM	0	0		
LUT as Logic	55	0	41000	0.13
using O5 output only	0			
using O6 output only	29			
using O5 and O6	26			
LUT as Memory	0	0	13400	0.00
LUT as Distributed RAM	0	0		
LUT as Shift Register	0	0		
LUT Flip Flop Pairs	0	0	41000	0.00
Unique Control Sets	0			

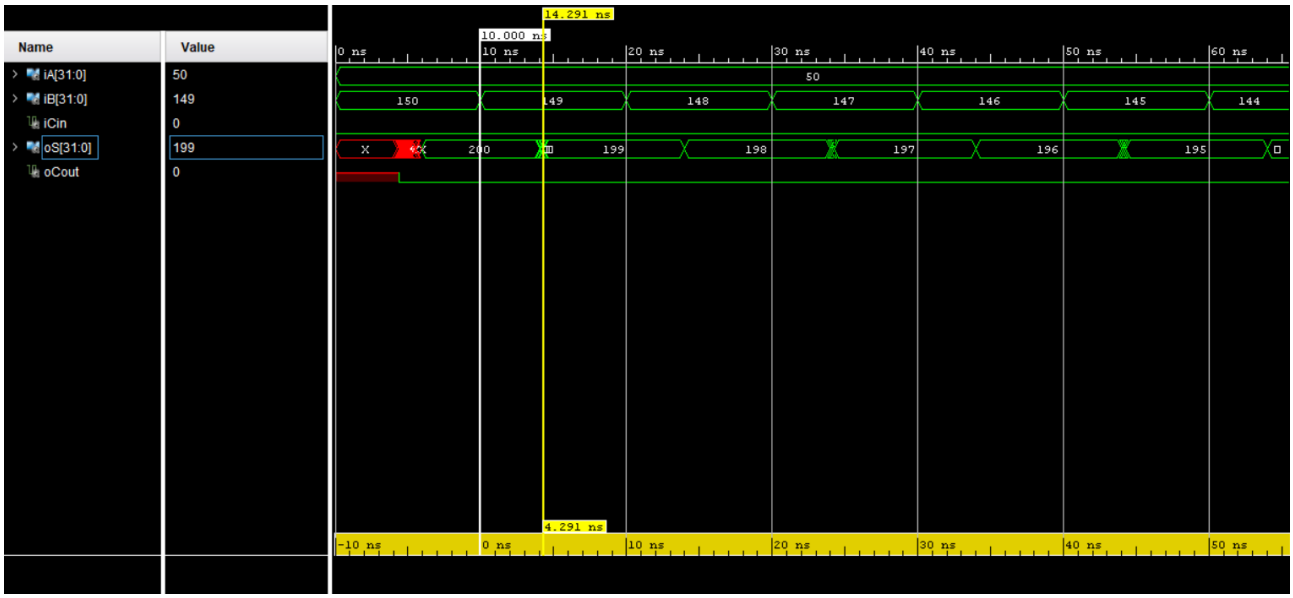
Da qui vediamo anche come abbiamo usato le varie LUT, 29 per memorizzare una sola tabella di verità (una sola uscita), le altre 26 invece sono state usate per due funzioni logiche, infatti avranno due uscite.

Site Type	Used	Fixed	Available	Util%
Bonded IOB	98	0	300	32.67
IOB Master Pads	46			
IOB Slave Pads	49			
Bonded IPADs	0	0	26	0.00
Bonded OPADs	0	0	16	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	0	0	6	0.00
IBUFDS	0	0	288	0.00
GTXE2_COMMON	0	0	2	0.00
GTXE2_CHANNEL	0	0	8	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	300	0.00
ODELAYE2/ODELAYE2_FINEDELAY	0	0	100	0.00
IBUFDS_GTE2	0	0	4	0.00
ILOGIC	0	0	300	0.00
OLOGIC	0	0	300	0.00

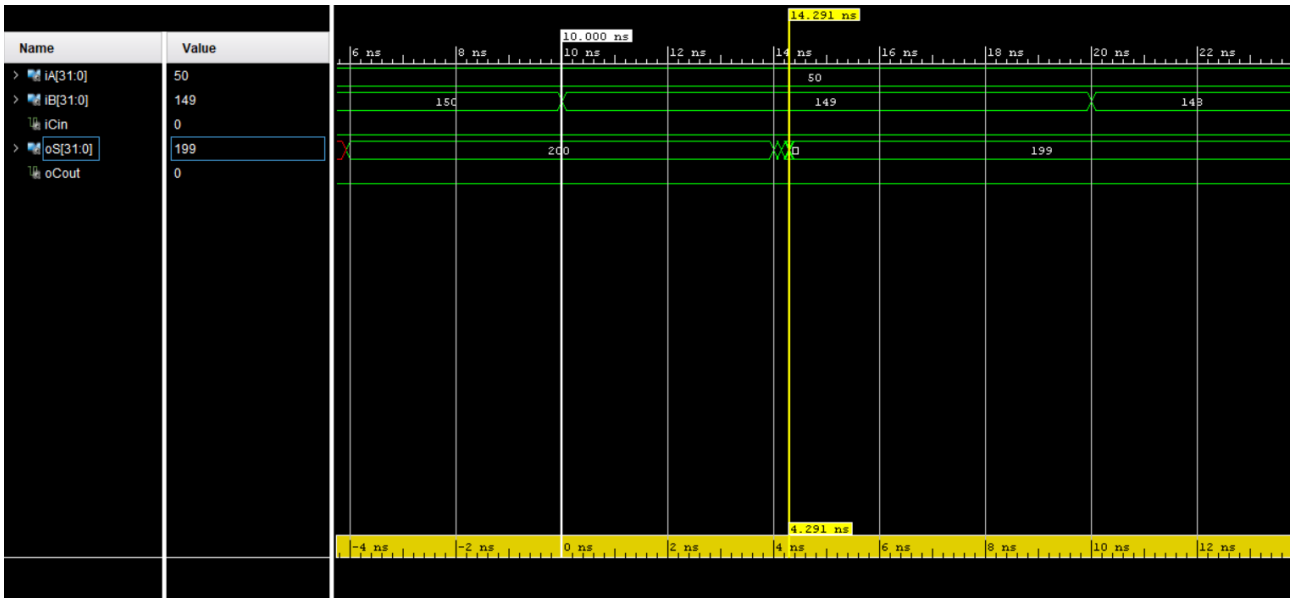
Qui invece troviamo le porte di ingresso ed uscita (miste) utilizzate nel nostro circuito, che in questo caso sono 98.

Ref Name	Used	Functional Category
IBUF	65	IO
OBUF	33	IO
LUT5	32	LUT
LUT3	21	LUT
LUT6	14	LUT
LUT4	14	LUT

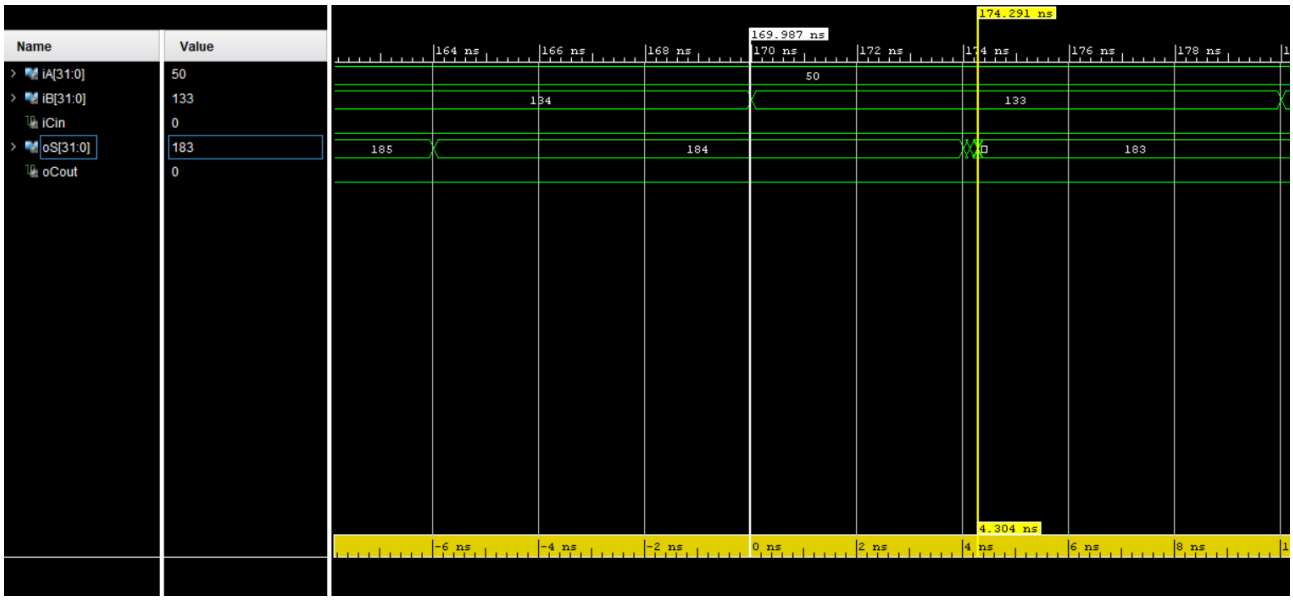
Qui troviamo un riassunto delle porte e delle LUT utilizzate. Possiamo eseguire un'altra simulazione, una volta eseguita l'implementazione, che sarebbe quella Post-Implementation Simulation, per completare totalmente anche la caratterizzazione del circuito in termini di ritardo:



Questo risulta essere la simulazione del circuito una volta implementato e notiamo come si conserva il "segno rosso" iniziale, sempre per gli stessi motivi precedenti, ma questa volta i tempi saranno cambiati:



Esaminando la stessa operazione precedente (50+149) notiamo che il tempo necessario a compere l'operazione è adesso di 4,291 ns; adesso il tempo è aumentato perché si tengono in considerazione i ritardi dei collegamenti, prima non avevamo questo ritardo perché, non avendo eseguito l'implementazione, non avevamo stabilito come le varie LUT sarebbero state collegate. Dunque, in questo caso bisognerà anche vedere ogni quanto mandare i segnali, (ad esempio con il nostro wait for), per evitare che nel tempo il circuito torni nella fase di indeterminazione per la sovrapposizione di bit che non riesce a calcolare. Dando qualche altro esempio di somme in questa fase con i relativi ritardi:



Dunque con i 10ns, riusciamo a contare tutti i ritardi dovuti alle varie interconnessioni con le LUT, grazie ai quali noteremo la differenza tra le due simulazioni post-synthesis e post-implementation.

Realizzato da:

Risafi Natale Francesco Pio

Matricola: 223491