



# SOMMATORE CA2 8,16,32 BIT



*-Rabbia Marco 220216*

*-Lazzaro Francesco 220810*

# TRACCIA PROGETTO

Si richiede di descrivere un circuito che riceve in ingresso due numeri in complemento a 2, A e B ad n-bit, e due segnali di controllo c1 e c0, in base ai quali si deve stabilire l'operazione da eseguire:

- se  $c1=c0=0$ ,  $A+B$ ;
- se  $c1=0$  e  $c0=1$ ,  $A-B$ ;
- se  $c1=1$  e  $c0=0$ ,  $-A+B$ ;
- se  $c1=c0=1$ ,  $-A-B$ .

Il circuito dev'essere pipeline e deve impiegare registri sensibili a i fronti di salita del segnale di clock. Si richiede, oltre alla verifica funzionale della struttura progettata, la sua caratterizzazione (valutazione di risorse occupate, massima frequenza di funzionamento, latenza e potenza dissipata) per  $n=8, 16$  e  $32$ .

# Cos'è un Registro?

I **registri** sono piccole aree di memoria all'interno di un computer che possono essere utilizzate rapidamente dal microprocessore per immagazzinare e manipolare i dati. Spesso, il termine "**registro**" si riferisce a un insieme di registri che possono essere indirizzati direttamente dalle istruzioni di input e output del microprocessore. Questi registri sono anche noti come "**architected registers**" e vengono utilizzati per effettuare calcoli e controllare il flusso delle istruzioni all'interno del computer.

I registri sono molto veloci nel manipolare i dati e sono solitamente misurati in base al numero di bit che possono contenere (ad esempio, 8-bit o 32-bit).

Le architetture dei computer moderne spesso adottano una struttura a **pipeline** che ottimizza l'utilizzo della memoria limitando l'accesso alle sole istruzioni load e store e utilizzando registri e costanti per l'esecuzione di tutte le altre istruzioni. Questo tipo di architettura, nota come **RISC (Reduced Instruction Set Computing)** o come architetture load-store, permette di ottenere vantaggi in termini di prestazioni e efficienza.

# Tipi di Registri



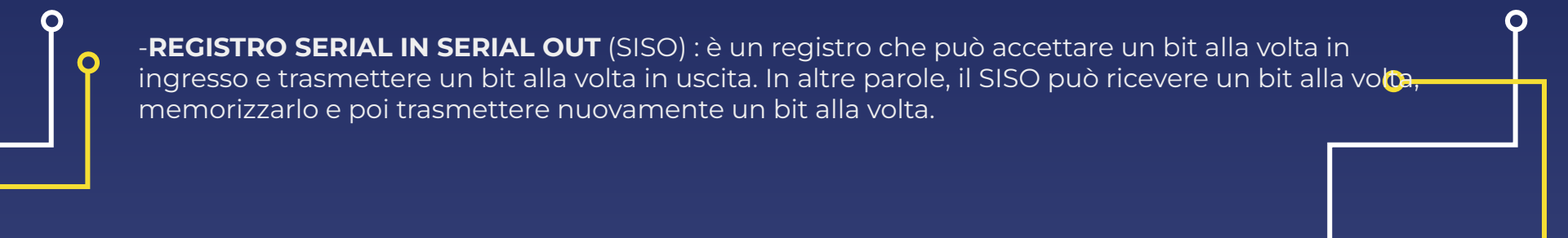
Esistono diversi tipi di registri che si differenziano in base all'input e all'output, tra questi ci sono:

-**REGISTRO PARALLEL IN PARALLEL OUT** (PIPO) : è un registro che può accettare una serie di bit in ingresso in modo parallelo e trasmetterli in uscita sempre in modo parallelo. In altre parole, il PIPO può ricevere una serie di bit tutti insieme, memorizzarli e poi trasmettere nuovamente tutti insieme (registro utilizzato per risolvere la traccia).

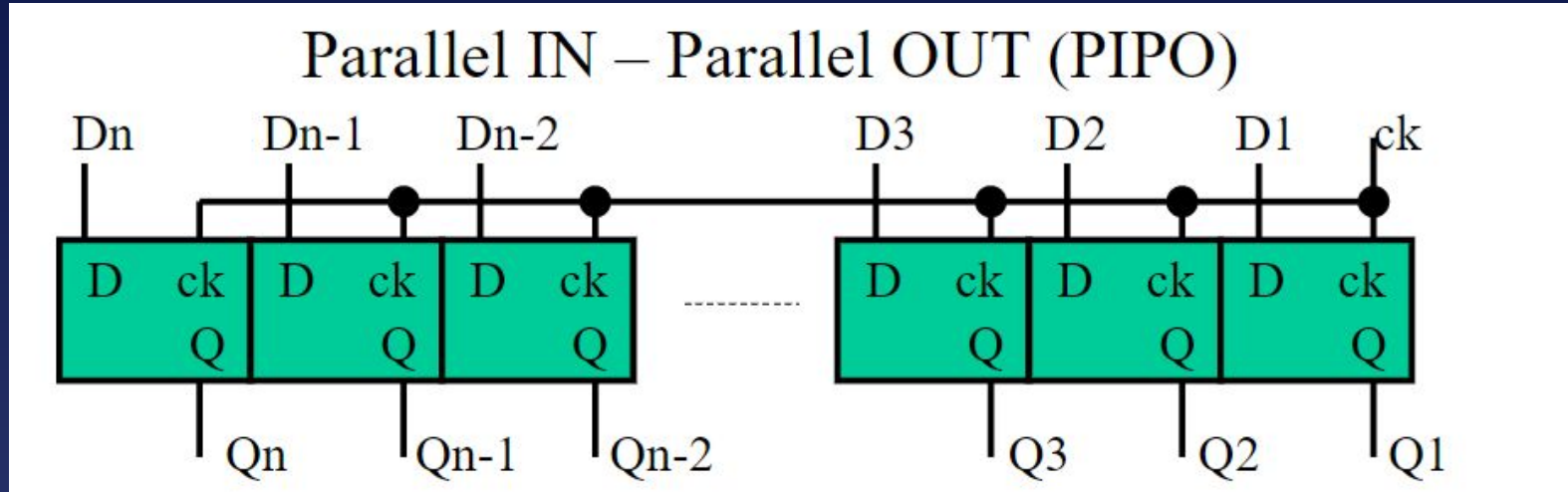
-**REGISTRO PARALLEL IN SERIAL OUT** (PISO) : è un registro che può accettare una serie di bit in ingresso in modo parallelo e trasmettere un bit alla volta in uscita. In altre parole, il PISO può ricevere una serie di bit tutti insieme, memorizzarli e poi trasmettere un bit alla volta.

-**REGISTRO SERIAL IN PARALLEL OUT** (SIPO) : è un registro che può accettare un bit alla volta in ingresso e trasmettere una serie di bit in uscita in modo parallelo. In altre parole, il SIPO può ricevere un bit alla volta, memorizzarli e poi trasmettere nuovamente tutti insieme.

-**REGISTRO SERIAL IN SERIAL OUT** (SISO) : è un registro che può accettare un bit alla volta in ingresso e trasmettere un bit alla volta in uscita. In altre parole, il SISO può ricevere un bit alla volta, memorizzarlo e poi trasmettere nuovamente un bit alla volta.



# Registro Parallel In Parallel Out



# Codice VHDL Registri

Questo codice VHDL descrive un registro generico a n bit. L'entity ha quattro porte: **D**, **Clk**, **Reset** e **Q**.

L'architecture "**myReg**" del registro include un process che viene eseguito ogni volta che si verifica un rising edge sul segnale di clock Clk. All'interno del process, viene effettuato un controllo sulla porta Reset. Se il segnale Reset è uguale a 1, il vettore di uscita Q viene impostato su una stringa di n bit tutti uguali a 0. Altrimenti, il vettore di uscita Q viene impostato sui dati presenti sulla porta D.

In questo modo, il registro viene aggiornato ogni volta che si verifica un rising edge sul segnale di clock Clk, a meno che il segnale Reset non sia impostato su 1, in cui caso il registro viene resettato a zero.

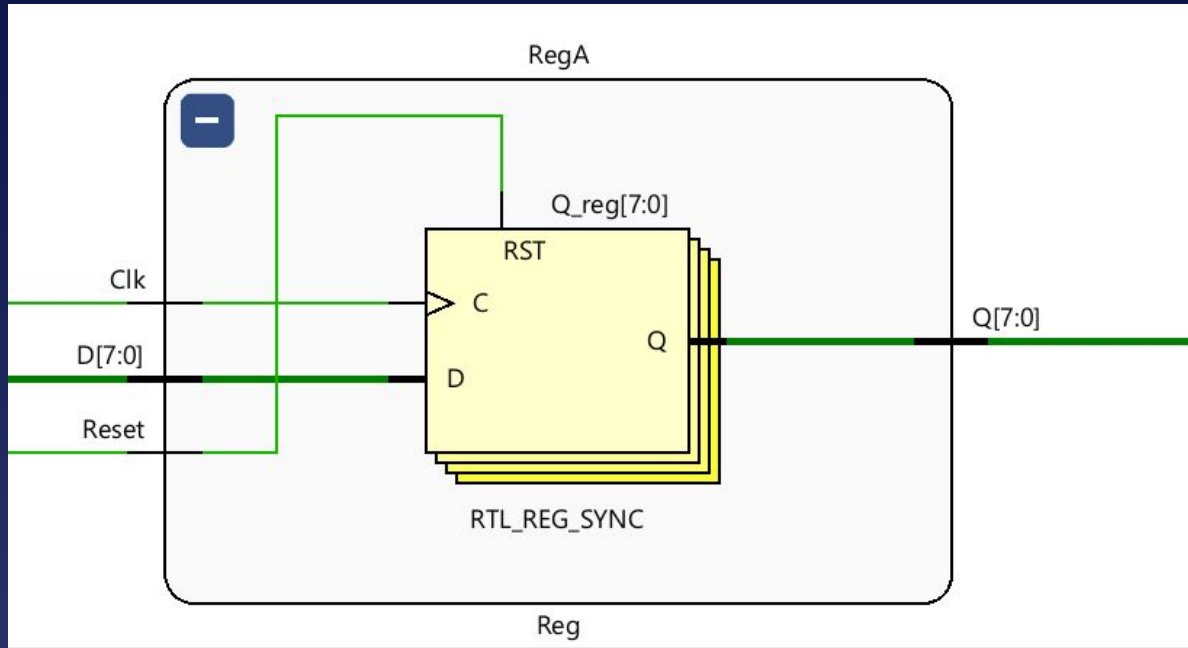
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg is
generic (n:integer);
  Port ( D : in STD_LOGIC_VECTOR (n-1 downto 0);
        Clk : in STD_LOGIC;
        Reset : in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR (n-1 downto 0));
end Reg;

architecture myReg of Reg is

begin
  process (Clk) begin
    if rising_edge(Clk) then
      if (Reset = '1') then Q<= (others =>'0');
      else Q<=D;
      end if;
    end if;
  end process;
end myReg;
```

# Schematic



# Cosa si intende per Pipeline

Le pipeline sono una tecnica di elaborazione utilizzata in informatica per aumentare l'efficienza e le prestazioni di un sistema di elaborazione. In una pipeline, un'attività viene suddivisa in piccole **fasi** o "**stadi**", ognuno dei quali viene eseguito da un componente hardware o software dedicato. Ciascuno di questi componenti lavora indipendentemente su una porzione di dati alla volta, in modo che il sistema possa elaborare più dati contemporaneamente.

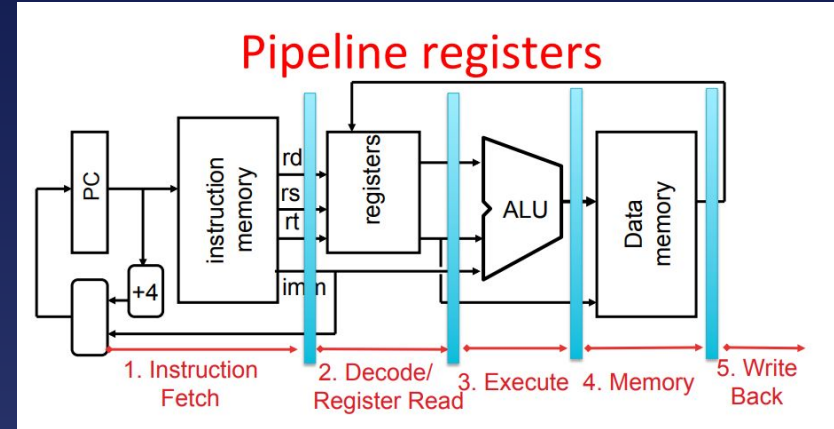
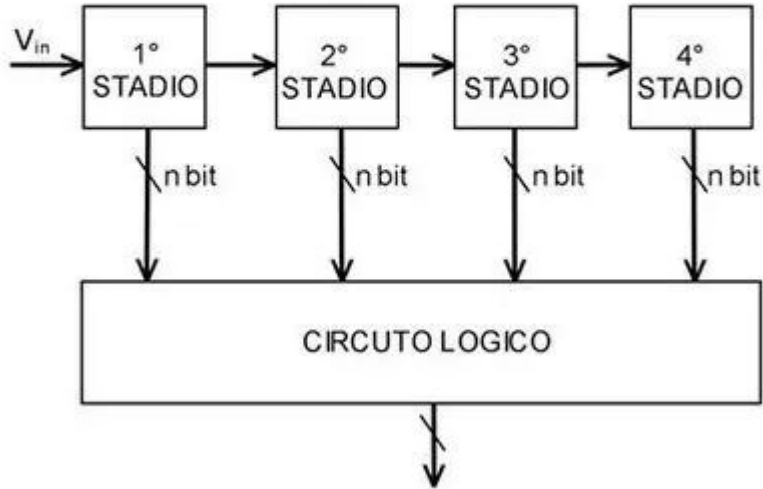
Ad esempio, in una pipeline di elaborazione di istruzioni, il primo stadio potrebbe essere quello di decodificare l'istruzione, il secondo stadio potrebbe essere quello di recuperare i dati richiesti dall'istruzione, il terzo stadio potrebbe essere quello di eseguire l'operazione richiesta dall'istruzione e così via. Mentre un'istruzione viene elaborata attraverso ciascuno di questi stadi, il sistema può iniziare a elaborare un'altra istruzione attraverso lo stesso insieme di stadi.

**Le pipeline** sono utilizzate in molti sistemi di elaborazione, inclusi i microprocessori dei computer, gli acceleratori di rete e i sistemi di database. **Consentono di** aumentare l'efficienza del sistema, poiché ogni stadio può **lavorare in parallelo su una porzione di dati**, riducendo così i tempi di attesa. Tuttavia, le pipeline presentano anche alcune sfide, come la gestione della sincronizzazione tra gli stadi e il rischio di collisioni tra le diverse attività in esecuzione.



# Schema di funzionamento di una Pipeline

## *Schema Funzionamento di una Pipeline*



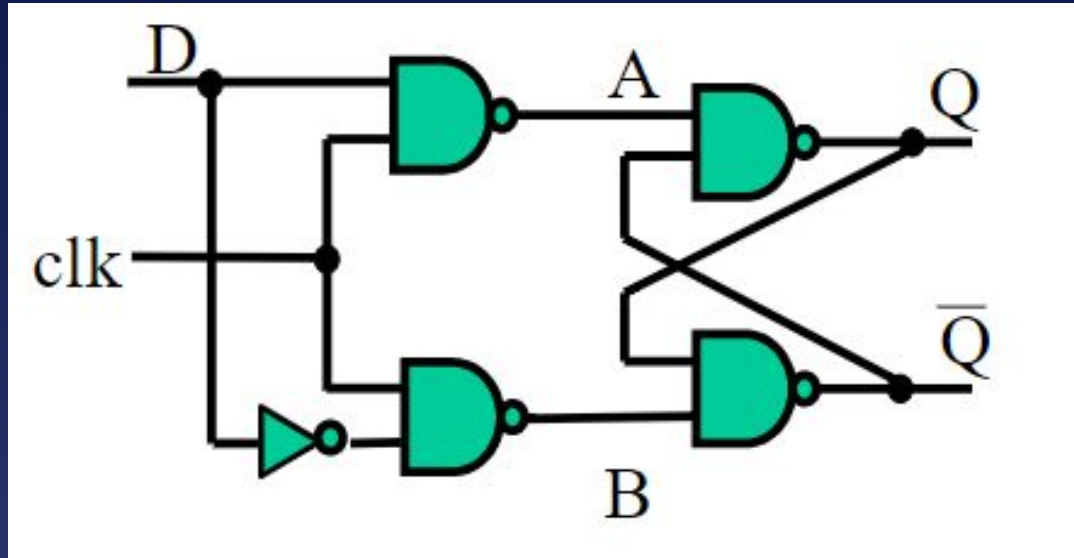
# Cosa sono i Flip-Flop

In informatica, un **flip-flop** è un tipo di circuito a stato indeterminato che può essere utilizzato per immagazzinare un bit di dati. Un flip-flop è costituito da due stati stabili, noti come "1" e "0", e può essere impostato in uno dei due stati in base a un segnale di ingresso. Una volta impostato in uno stato, il flip-flop rimarrà in quello stato finché non viene fornito un nuovo segnale di ingresso per cambiare lo stato.

I flip-flop sono stati utilizzati in passato come elementi di memoria all'interno dei computer, ma oggi vengono utilizzati principalmente per creare segnali di clock e per sincronizzare il flusso di dati all'interno di un sistema.

I registri sono simili ai flip-flop in quanto possono essere utilizzati per immagazzinare e manipolare i dati all'interno di un computer. Tuttavia, ci sono alcune differenze importanti tra i due. I registri sono solitamente più grandi dei flip-flop e possono contenere più bit di dati. Inoltre, i registri possono essere indirizzati direttamente dal microprocessore, mentre i flip-flop vengono utilizzati principalmente per creare segnali di clock e sincronizzare il flusso di dati. Infine, i registri sono utilizzati principalmente per effettuare calcoli e controllare il flusso delle istruzioni all'interno di un computer, mentre i flip-flop vengono utilizzati principalmente per immagazzinare dati a breve termine.

# Schema di funzionamento di un Flip-Flop



# Codice VHDL Flip-Flop

Il codice VHDL per l'implementazione dei flip-flop è lo stesso dei registri in quanto i registri possono essere effettivamente composti da un insieme di flip-flop collegati tra loro. In passato infatti una delle prime implementazioni dei registri di dati avveniva proprio tramite utilizzo di flip-flop.

In questo caso il generico  $n$  è uguale ad 1, pertanto viene visto come un Reg di dimensione 1, 0 downto 0.

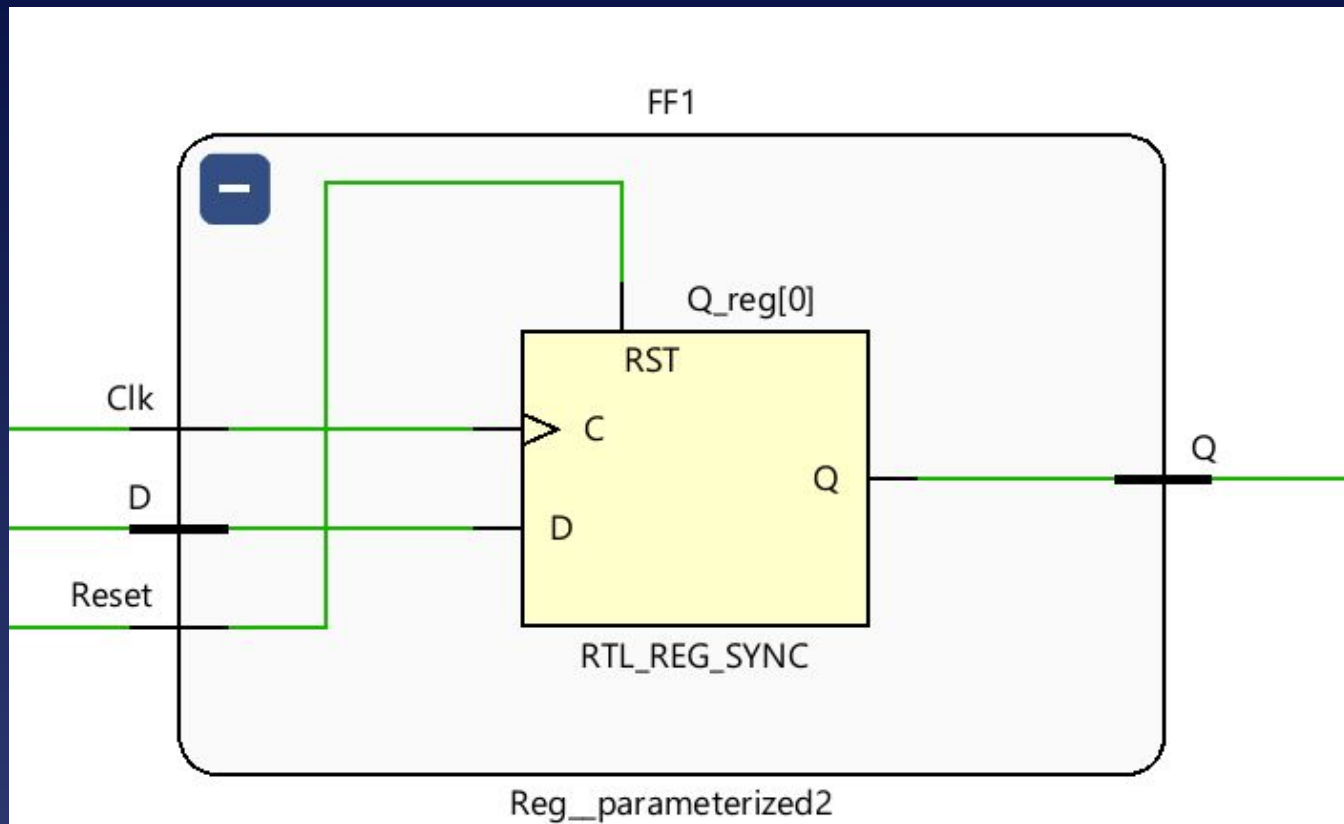
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg is
generic (n:integer);
  Port ( D : in STD_LOGIC_VECTOR (n-1 downto 0);
        Clk : in STD_LOGIC;
        Reset : in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR (n-1 downto 0));
end Reg;

architecture myReg of Reg is

begin
  process (Clk) begin
    if rising_edge(Clk) then
      if (Reset = '1') then Q<= (others =>'0');
      else Q<=D;
      end if;
    end if;
  end process;
end myReg;
```

# Schematic



# Implementazione behavioural del RCA

Il comportamento del report carry adder può essere descritto utilizzando i segnali di **Propagate** (P) e **Generate** (G) e le relative equazioni delle uscite e dei riporti.

Il segnale di Propagate (P) viene generato quando entrambi i bit di ingresso sono 1, mentre il segnale di Generate (G) viene generato quando almeno uno dei due bit di ingresso è 1.

Le equazioni delle uscite per il report carry adder sono:

Somma (S) = **A XOR B XOR Cin** (carry in)

Carry out (Cout) = **(A AND B) OR (Cin AND (A XOR B))**

Le equazioni del riporto per il report carry adder sono:

**G = A AND B**

**P = A XOR B**

Il comportamento del report carry adder può quindi essere descritto come segue:

- Se entrambi i bit di ingresso sono 1, il segnale di Propagate (P) viene generato e il bit di uscita Carry out (Cout) viene impostato su 1.
- Se almeno uno dei due bit di ingresso è 1, il segnale di Generate (G) viene generato e il bit di uscita Carry out (Cout) viene impostato su 1 se il bit di ingresso Carry in (Cin) è anche 1, altrimenti viene impostato su 0.
- Se nessuno dei due bit di ingresso è 1, né il segnale di Propagate né quello di Generate vengono generati e il bit di uscita Carry out (Cout) viene impostato su 0.

```

c(0) <= Contr1(0) xor Contr2(0);

p(n) <= MA(n-1) xor MB(n-1);
g(n) <= MA(n-1) and MB(n-1);

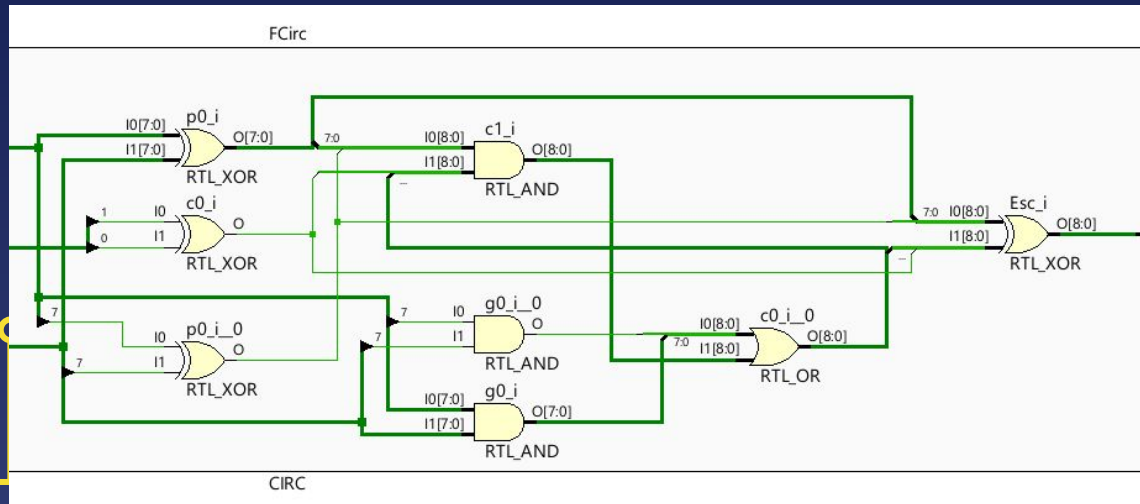
p(n-1 downto 0) <= MA xor MB;
g(n-1 downto 0) <= MA and MB;

c(n+1 downto 1) <= g or (p and c(n downto 0));
Esc <= p xor c(n downto 0);

end myCIRC;

```

## CODICE VHDL RCA BEHAVIOURAL



## SCHEMATIC



# CIRC: cosa avviene nel circuito combinatorio

Le porte A e B sono entrambi vettori che rappresentano i dati di ingresso del circuito. La porta **Contr1** è un vettore di lunghezza 1, che viene utilizzato per controllare il comportamento del circuito sui dati di ingresso A. La porta **Contr2** è un vettore di lunghezza 1, che viene utilizzato per controllare il comportamento del circuito su B. La porta **Esc** è un vettore di lunghezza n+1, che rappresenta i dati di uscita del circuito.

Questo codice VHDL descrive un circuito che esegue l'operazione di addizione tra due valori binari A e B. I due segnali di controllo, "Contr1" e "Contr2", determinano se A e B vanno invertiti prima dell'addizione, attraverso l'uso di due MUX.

Inoltre, se entrambi i segnali di controllo sono impostati su 1, viene eseguita l'operazione di sottrazione di B dall'opposto di A. Per fare ciò, viene utilizzato un sommatore **RCA behavioural** ausiliario per calcolare l'equivalente negativo di B, ovvero "**not B + 1**", e quindi viene eseguita l'addizione di "**notA**" e "**notB + 1**" utilizzando il sommatore principale.

Il risultato finale viene ottenuto sommando ulteriormente 1 attraverso un altro sommatore,, ovvero "**Esc + 1**".



# Codice VHDL CIRC



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

entity CIRC is
generic (n: integer);
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
          B : in STD_LOGIC_VECTOR (n-1 downto 0);
          Contr1 : in STD_LOGIC_VECTOR (0 downto 0);
          Contr2 : in STD_LOGIC_VECTOR (0 downto 0);
          Esc : out STD_LOGIC_VECTOR (n downto 0));
end CIRC;

architecture myCIRC of CIRC is

--segnali ausiliari per A/notA e B/notB
signal MA,MB :std_logic_vector (n-1 downto 0);

--segnali per il sommatore principale
signal p,g :std_logic_vector (n downto 0);
signal C :std_logic_vector (n+1 downto 0);

--segnali per il sommatore ausiliario
signal EX,zeros: std_logic_vector (n downto 0);
signal p2,g2 :std_logic_vector (n+1 downto 0);
signal C2 :std_logic_vector (n+2 downto 0);
signal Esc2:std_logic_vector(n+1 downto 0);
```

```
begin
    with Contr2 select
        MB <= B when "0",
            not(B) when "1",
            (others => 'X') when others;
    with Contr1 select
        MA <= A when "0",
            not(a) when "1",
            (others => 'X') when others;

    C(0)<= Contr1(0) or Contr2(0);

    process begin

        if (Contr1(0)='1')and(Contr2(0)='1') then
            -- Operazione -A-B = notA + notB + 1 + 1 =>
            -- notA + notB + 1 viene inserito nel sommatore principale =>

            --primo sommatore => notA + notB + 1

            p(n)<=MA(n-1) xor MB(n-1);
            g(n)<=MA(n-1) and MB(n-1);

            p(n-1 downto 0)<=MA xor MB;
            g(n-1 downto 0)<=MA and MB;

            C(n+1 downto 1)<= g or (p and C(n downto 0));
            EX <= p xor C(n downto 0);
```

```
--sommatore ausiliario => (notA + notB + 1) + 1

        zeros<=(others=>'0');
        C2(0)<='1';
        p2(n+1)<=EX(n) xor zeros(n);
        g2(n+1)<=EX(n) and zeros(n);

        p2(n downto 0)<=EX xor zeros;
        g2(n downto 0)<=EX and zeros;

        C2(n+2 downto 1)<= g2 or (p2 and C2(n+1 downto 0));
        Esc <= p2(n downto 0) xor C2(n downto 0);

    else
        p(n)<=MA(n-1) xor MB(n-1);
        g(n)<=MA(n-1) and MB(n-1);

        p(n-1 downto 0)<=MA xor MB;
        g(n-1 downto 0)<=MA and MB;

        C(n+1 downto 1)<= g or (p and C(n downto 0));
        Esc <= p xor C(n downto 0);

    end if;
    wait for 0.1ns;
end process;

end myCIRC;
```

# Testbench registro a 8 bit

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

entity TestBench is
end TestBench;

architecture MySim of TestBench is

    component Main

        Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
              B : in STD_LOGIC_VECTOR (7 downto 0);
              Contr1 : in STD_LOGIC_VECTOR (0 downto 0);
              Contr2 : in STD_LOGIC_VECTOR (0 downto 0);
              Clk, Reset : in STD_LOGIC;
              Esc : out STD_LOGIC_VECTOR (8 downto 0));
    end component;

    signal IA,IB: std_logic_vector(7 downto 0);
    signal IC1 : STD_LOGIC_VECTOR (0 downto 0);
    signal IC2 : STD_LOGIC_VECTOR (0 downto 0);
    signal IClk: STD_LOGIC := '0';
    signal IReset: STD_LOGIC := '0';
    signal OEsc: STD_LOGIC_VECTOR (8 downto 0);
    signal Tclk: time :=10 ns;
```

```
begin

    CUT: Main port map (IA,IB,IC1,IC2,IClk,IReset,OEsc);

    process begin
        wait for Tclk/2;
        IClk <= not IClk;
    end process;

    process begin
        for z in 0 to 1 loop
            IC1<=conv_std_logic_vector(z,1);

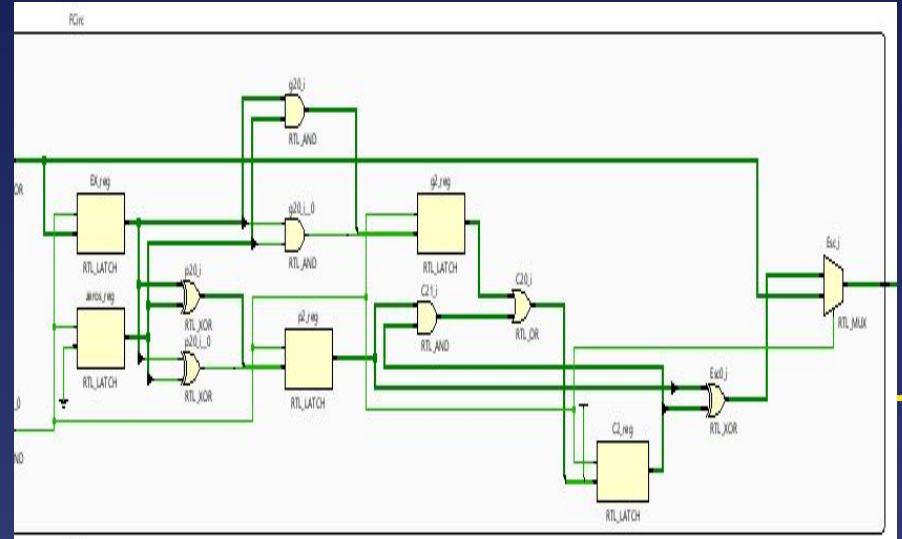
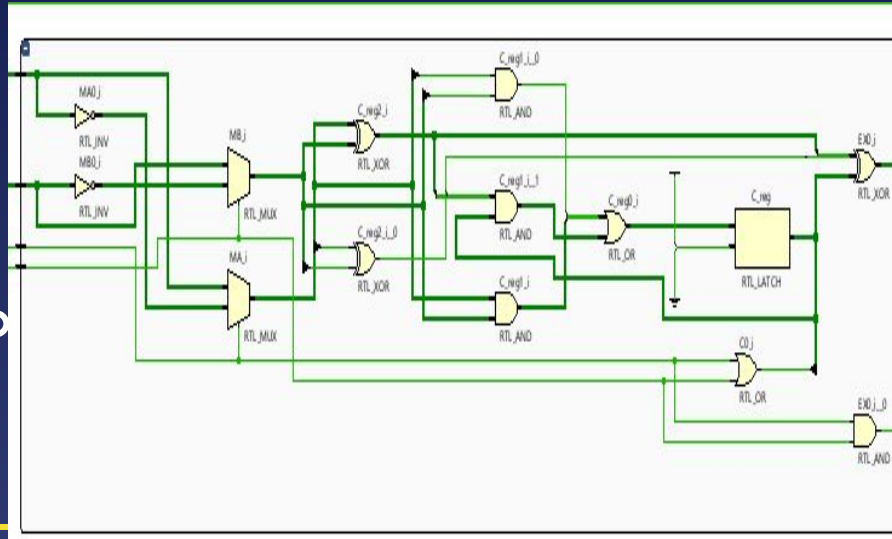
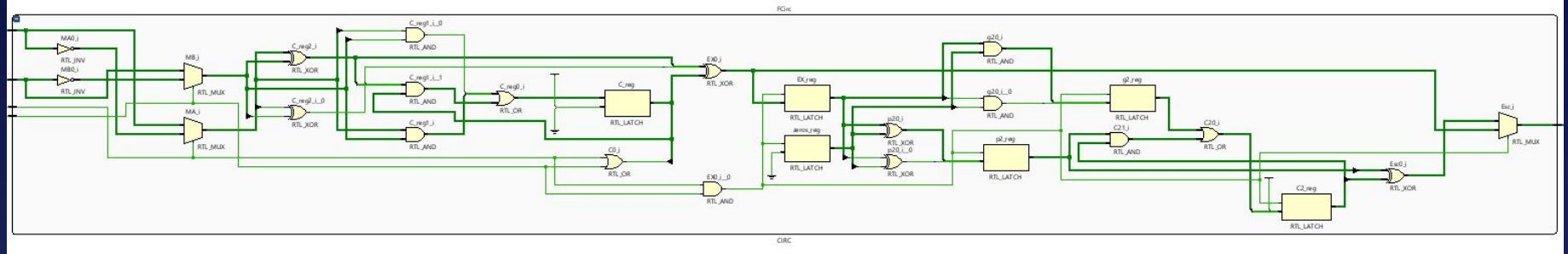
            for h in 0 to 1 loop
                IC2<=conv_std_logic_vector(h,1);

                for i in -5 to 5 loop
                    for j in -5 to 5 loop
                        IA<= conv_std_logic_vector(i,8);
                        IB<= conv_std_logic_vector(j,8);
                        wait for Tclk;
                    end loop;
                end loop;
            end loop;
        end loop;

    end process;

end MySim;
```

# Schematic



# Main: cosa fa e codice VHDL

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

entity Main is
generic (n: integer:=8);
  Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
        B : in STD_LOGIC_VECTOR (n-1 downto 0);
        Contr1 : in STD_LOGIC_VECTOR (0 downto 0);
        Contr2 : in STD_LOGIC_VECTOR (0 downto 0);
        Clk, Reset : in STD_LOGIC;
        Esc : out STD_LOGIC_VECTOR (n downto 0));
end Main;

architecture myMain of Main is

signal RA,RB: std_logic_vector(n-1 downto 0);
signal Rc1,Rc2: std_logic_vector(0 downto 0);
signal Ris: std_logic_vector(n downto 0);

component Reg
  generic (n: integer);
  Port ( D : in STD_LOGIC_VECTOR (n-1 downto 0);
        Clk : in STD_LOGIC;
        Reset : in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR (n-1 downto 0));
end component;
```

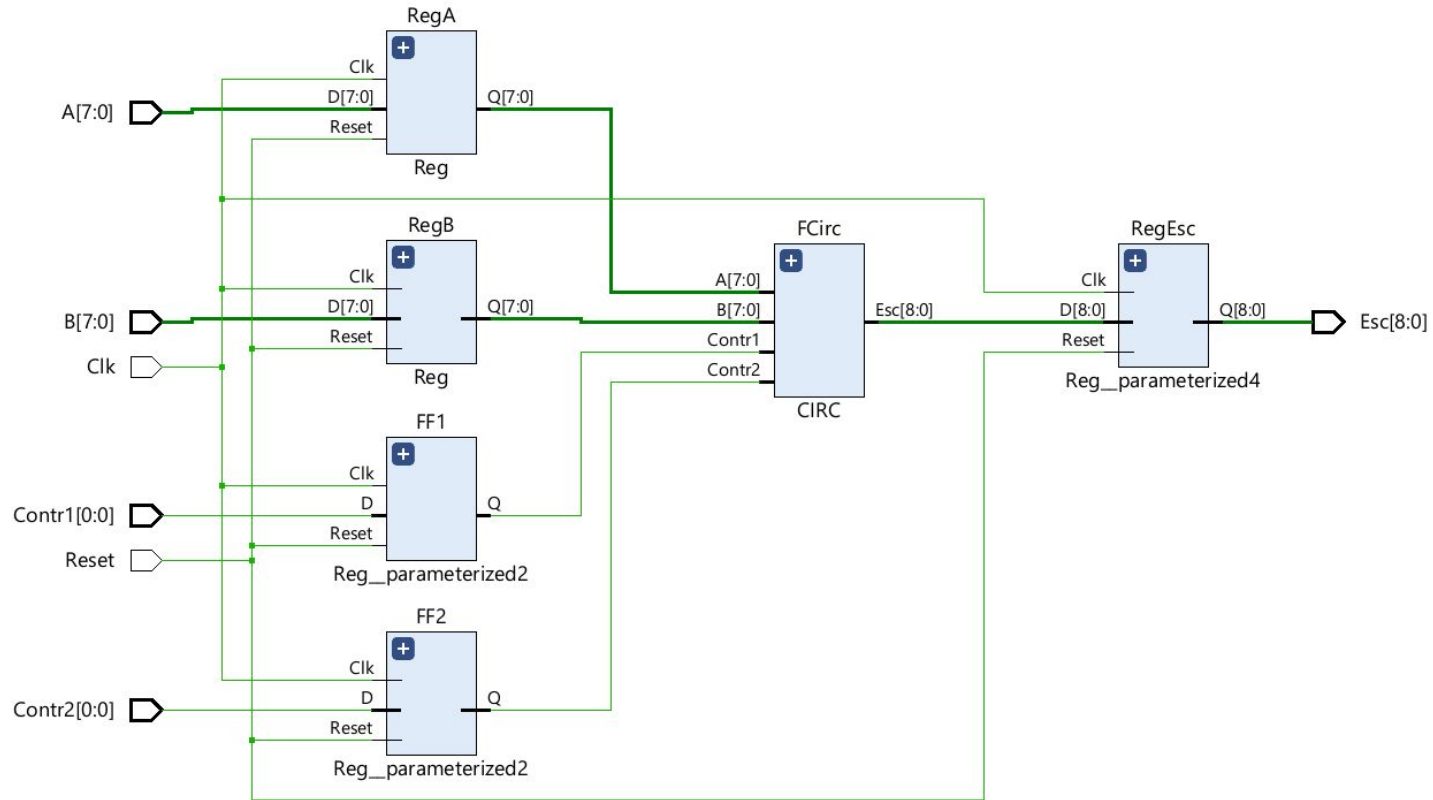
All'interno della sezione Main vengono implementati e inizializzati come component Reg, che serve per creare i registri e i flip-flop, e CIRC, il circuito combinatorio.

Il generico “n” in questo caso viene impostato a 8 e tale valore verrà propagato agli altri codici attraverso l'utilizzo dello statement generic map.

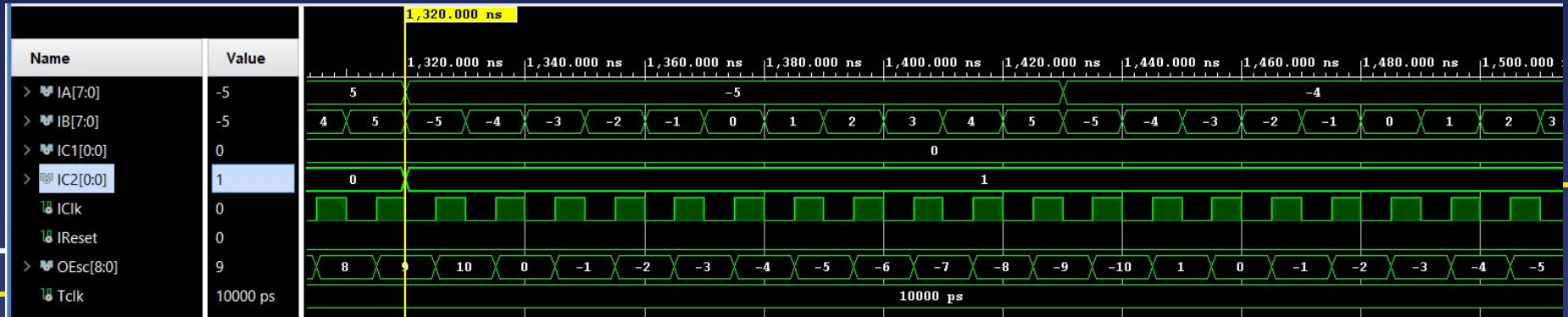
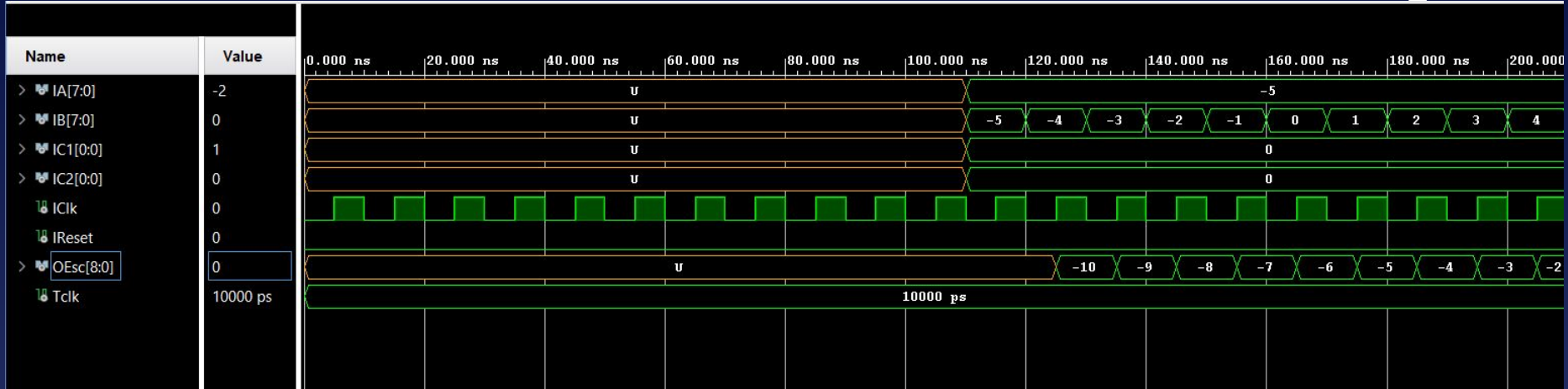
```
component CIRC
generic (n: integer);
  Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
        B : in STD_LOGIC_VECTOR (n-1 downto 0);
        Contr1 : in STD_LOGIC_VECTOR (0 downto 0);
        Contr2 : in STD_LOGIC_VECTOR (0 downto 0);
        Esc : out STD_LOGIC_VECTOR (n downto 0));
end component;

begin
  RegA: Reg generic map(n) port map(A, Clk, Reset, RA);
  RegB: Reg generic map(n) port map(B, Clk, Reset, RB);
  FF1: Reg generic map(1) port map(Contr1, Clk, Reset, Rc1);
  FF2: Reg generic map(1) port map(Contr2, Clk, Reset, Rc2);
  FCirc: CIRC generic map(n) port map(RA, RB, Rc1, Rc2, Ris);
  RegEsc: Reg generic map(n+1) port map(Ris, Clk, Reset, Esc);
end myMain;
```

# Schematic

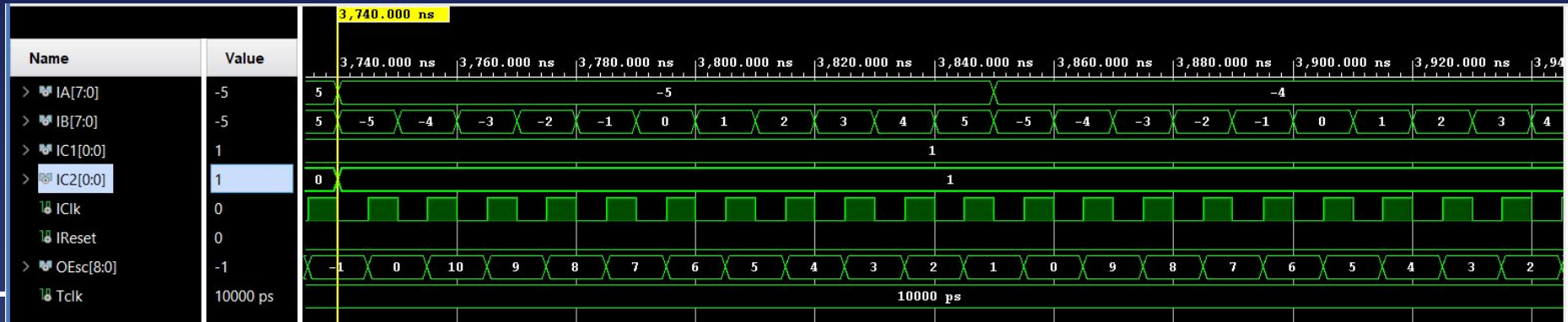
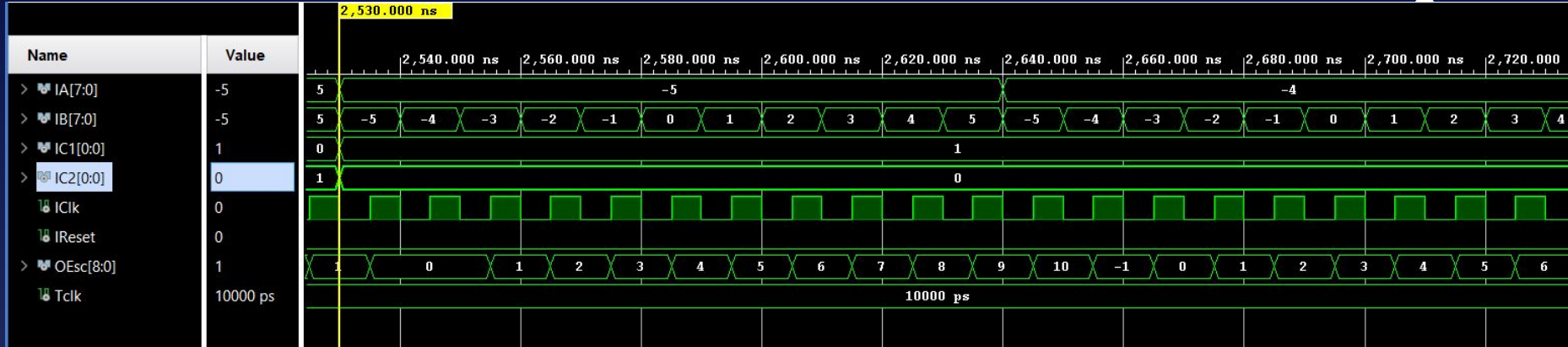


# Simulazione behavioural (00 e 01)



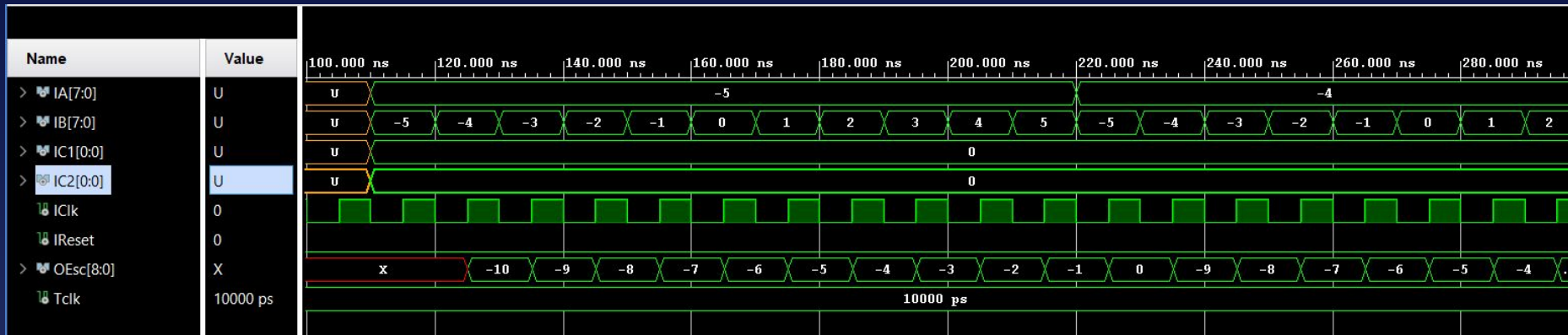


# Simulazione behavioural (10 e 11)



# Simulazione Post-Synthesis

FUNCTIONAL



TIMING





# Confronto simulazione Behavioral e Post-Synthesis

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.

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

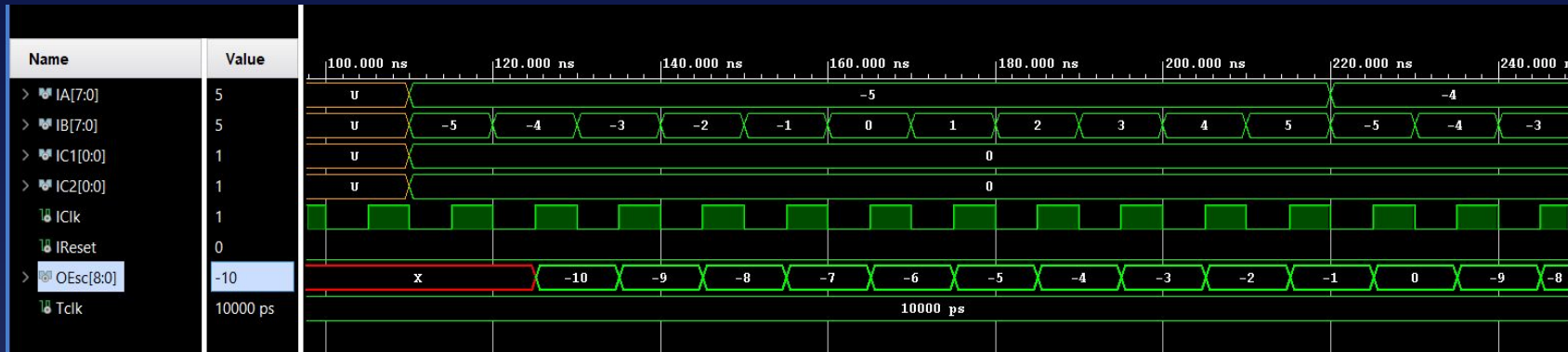
Per concludere entrambi i cicli for, in modo che vengano svolti tutti i casi possibili con i bit di controllo, il circuito impiega 4955 ns nella simulazione post sintesi functional, mentre nella timing 4958,742 ns, con un ritardo di 3,742 secondi.

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

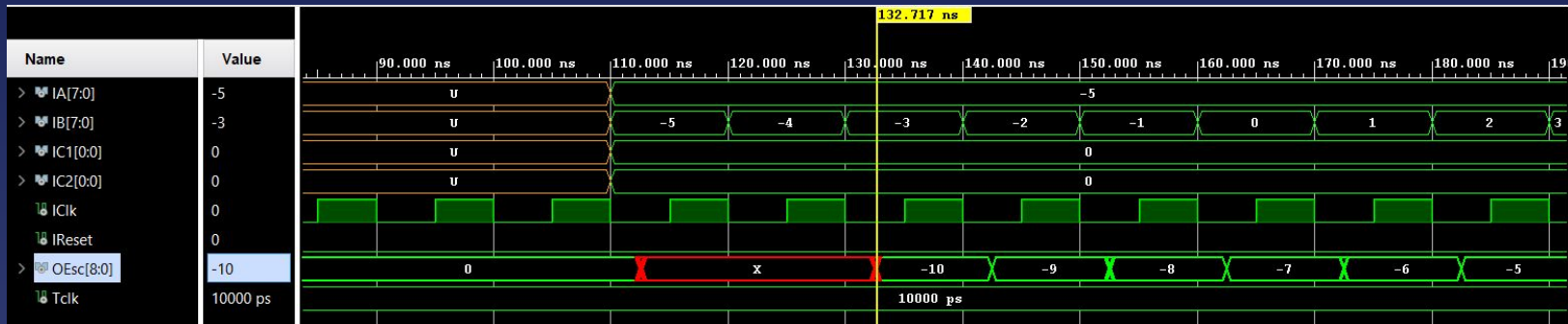
E' importante dunque passare ad una **implementazione del codice** ed a una **simulazione post-implementation**.

# Simulazione Post-Implementation

FUNCTIONAL



TIMING

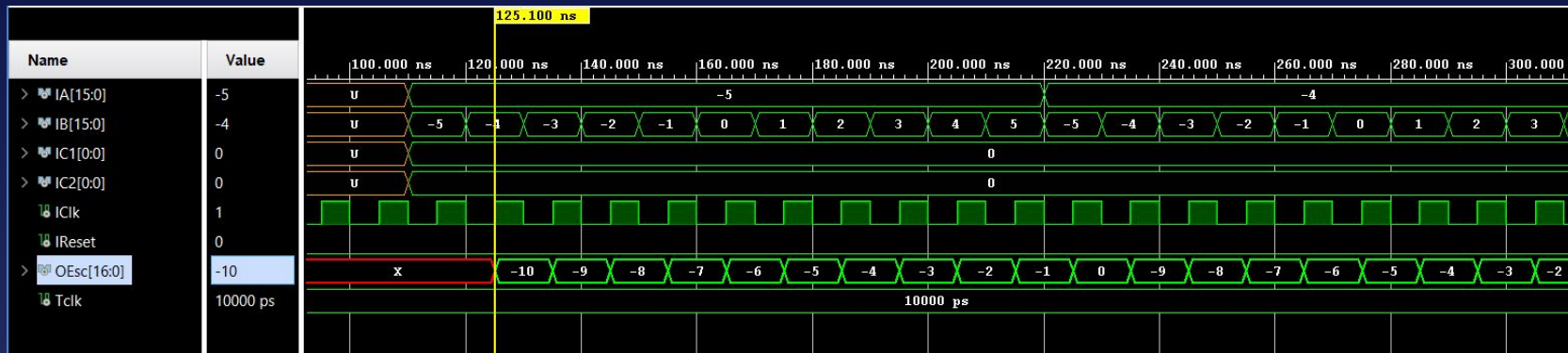


# Confronto simulazione Post-Synthesis e Post-Implementation

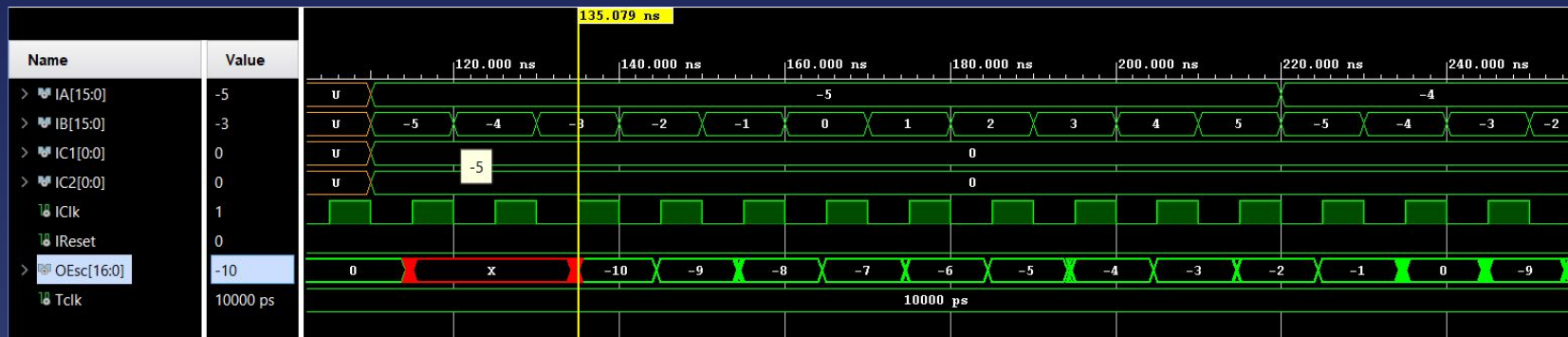
Nella simulazione **post-implementation** si può notare come il tempo impiegato rimane invariato nella simulazione **functional** 4955 ns ma vi è un'evidente discrepanza nel ritardo iniziale nella simulazione **timing** che risulta essere di 7,617 ns (a fronte dei 3,742 ns nella post-sintesi) facendo completare il circuito in 4962,617 ns.

# Simulazione Post-Implementation Registro a 16 bit

FUNCTIONAL

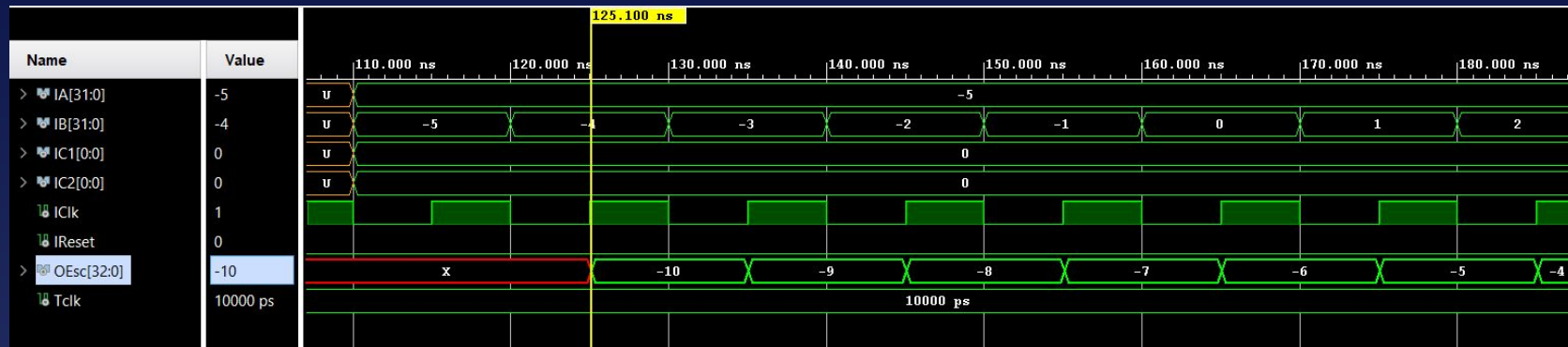


TIMING

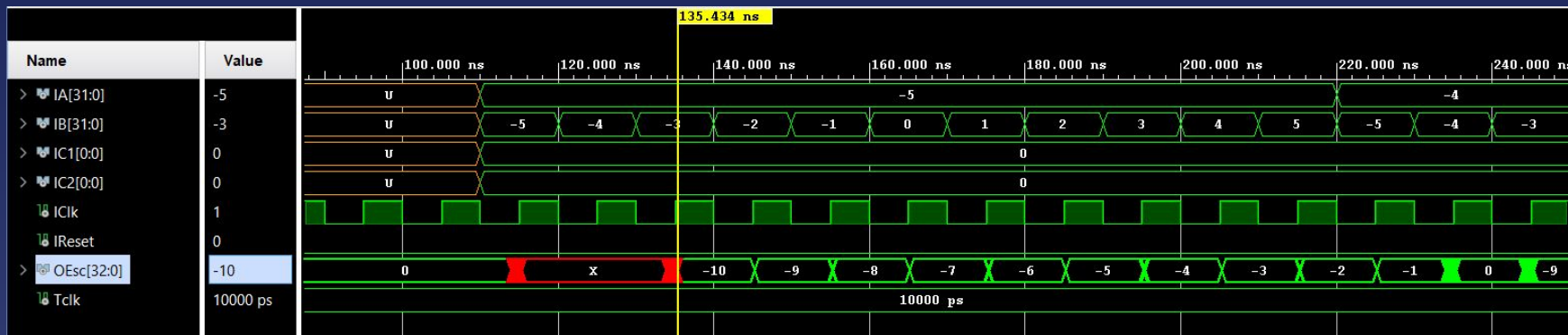


# Simulazione Post-Implementation Registro a 32 bit

FUNCTIONAL



TIMING



# Differenze tra simulazioni 8-16-32 bit

Per quanto riguarda le simulazioni post-implementation functional, in tutti e 3 i casi il tempo impiegato per aggiornare il primo valore del registro somma è di 15 ns, mentre il tempo impiegato per eseguire i cicli della simulazione è di 4955 ns.

Nelle simulazioni post-implementation timing invece si possono notare alcune differenze:

- 8 bit:            primo ritardo = 7,617 ns                      tempo di esecuzione finale = 4962,617 ns
- 16 bit:           primo ritardo = 9,210 ns                      tempo di esecuzione finale = 4964,533 ns
- 32 bit:           primo ritardo = 10,334 ns                      tempo di esecuzione finale = 4974,585 ns

# Constraints utilizzati

I **timing constraints** sono specifiche che definiscono i limiti di tempo entro cui un circuito o un sistema di elaborazione deve operare in modo corretto. Si tratta di una parte importante della progettazione di sistemi digitali, in quanto consentono di garantire che il sistema funzioni correttamente e in modo sincrono a livello di clock. I timing constraints possono essere utilizzati per specificare i requisiti di latenza per determinate operazioni, nonché per definire i limiti di frequenza di clock del sistema. In seguito sono riportati i timing constraints utilizzati rispettivamente dai circuiti 8,16 e 32 bit:

```
create_clock -period 3.00 -name Clokk -waveform {0.000 1.500} [get_nets Clk]
```

```
create_clock -period 5.00 -name Clokk -waveform {0.000 2.500} [get_nets Clk]
```

```
create_clock -period 7.200 -name Clokk -waveform {0.000 3.600} [get_nets Clk]
```

# Power Report con registri 8-16-32 bit

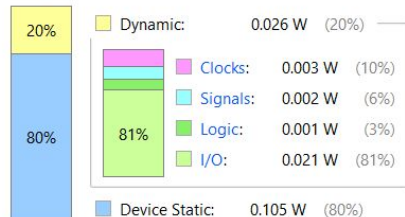
## Summary

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

**Total On-Chip Power:** 0.131 W  
**Design Power Budget:** Not Specified  
**Power Budget Margin:** N/A  
**Junction Temperature:** 26,5°C  
Thermal Margin: 58,5°C (4,9 W)  
Effective  $\theta_{JA}$ : 11,5°C/W  
Power supplied to off-chip devices: 0 W  
Confidence level: Low

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

### On-Chip Power



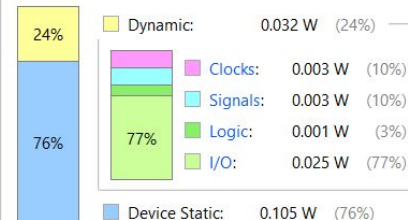
## Report con registri a 8 bit

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

**Total On-Chip Power:** 0.137 W  
**Design Power Budget:** Not Specified  
**Power Budget Margin:** N/A  
**Junction Temperature:** 26,6°C  
Thermal Margin: 58,4°C (4,9 W)  
Effective  $\theta_{JA}$ : 11,5°C/W  
Power supplied to off-chip devices: 0 W  
Confidence level: Low

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

### On-Chip Power



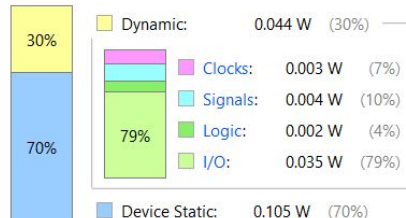
## Report con registri a 16 bit

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

**Total On-Chip Power:** 0.149 W  
**Design Power Budget:** Not Specified  
**Power Budget Margin:** N/A  
**Junction Temperature:** 26,7°C  
Thermal Margin: 58,3°C (4,9 W)  
Effective  $\theta_{JA}$ : 11,5°C/W  
Power supplied to off-chip devices: 0 W  
Confidence level: Low

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

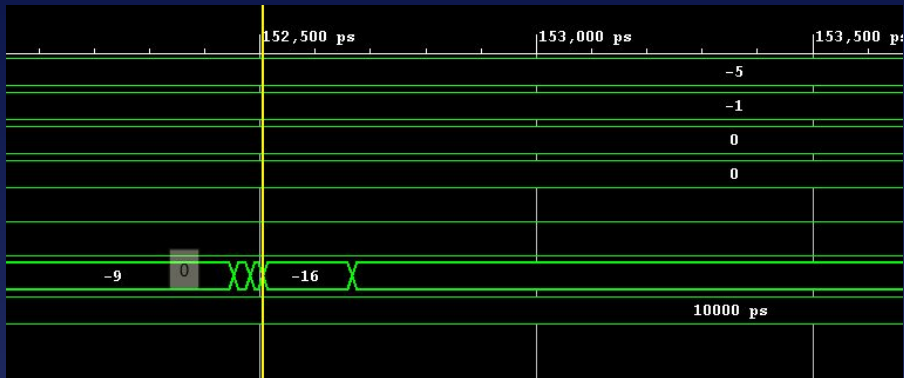
### On-Chip Power



## Report con registri a 32 bit



# Errori nelle simulazioni post- implementation timing



Durante le simulazioni, nel caso in cui i bit di controllo valgano "11" si riscontrano dei problemi nel risultato, dovuto molto probabilmente ai tempi del clock (che non sono stati sincronizzati in maniera corretta) che non permettono, in alcune iterazioni, di poter concludere in modo corretto.

Questi errori sono presenti in tutte e 3 le implementazioni.

