

GAIA
ASSUNTA
BERTOLINO
MAT. 209507

Carry-select a 16 bit

PROGETTO DI
ELETTRONICA DIGITALE

A.A.
2021/2022

INDICE

- Cos'è il carry-select
- Esempio di un Carry-select a 4 bit
- Multiplexer
- Full-adder
- Ripple-carry
- Carry-select
- Testbench e simulazione esaustiva
- Behavioral simulation
- Post-Synthesis simulation
- Post-Implementation simulation
- Report e approfondimento sui circuiti

COS'E' IL CARRY-SELECT

Un carry-select è un **circuito sommatore**. Esso permette di velocizzare il calcolo della somma fra due operandi introducendo della **ridondanza** nei calcoli.

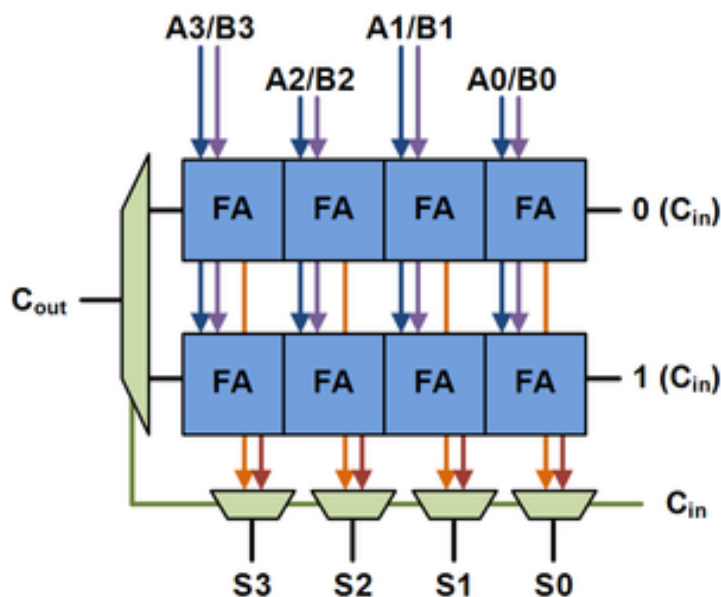
Infatti, assumendo di avere due operandi da n bit, il carry-select è costituito da p blocchi che elaborano m bit ciascuno in modo che $p \cdot m = n$. Tuttavia, questa non è l'unica metodologia di implementazione possibile e altre saranno discusse più avanti nella relazione.

Ogni blocco è a sua volta costituito da due ripple-carry adder che calcolano ciascuno il risultato della somma degli m bit associati considerando rispettivamente un riporto pari a uno e pari a zero. Si ottengono così due risultati e due riporti; i valori corretti vengono selezionati (da qui il nome select) attraverso un multiplexer associato.

In particolare, la scelta dei valori avviene quando il riporto del blocco precedente è stato calcolato in modo che faccia da segnale di controllo al multiplexer.

Tuttavia, esso si avvale di molti più calcoli rispetto ad un più semplice ma lento ripple-carry e ciò richiede all'incirca un 50% in più di risorse e dunque di **dissipazione di energia**.

ESEMPIO DI UN CARRY-SELECT A 4 BIT

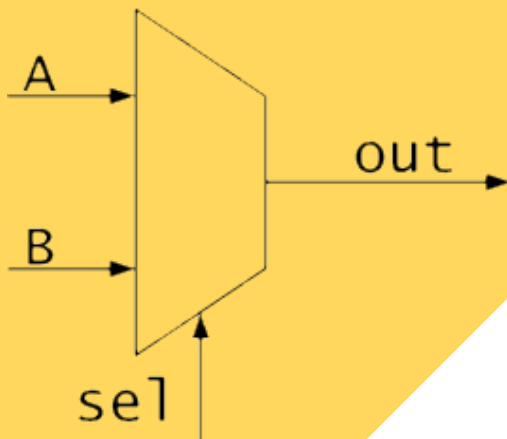


Vediamo ora nelle pagine a seguire i componenti citati e le loro descrizioni VHDL

Un carry-select a 4 bit si compone di due **ripple-carry**, ciascuno dei quali a sua volta composto da 4 full-adder. Infatti, ogni full-adder è capace di sommare due bit per volta; associando più 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 4 bit.

I riporti dei due ripple-carry sono collegati ad un multiplexer così come i risultati della somma dei singoli full-adder. Tali mux regolano quali siano i valori corretti (ovvero se quelli che tengono in considerazione un resto pari a 0 o pari ad 1) in base al valore del riporto in ingresso al ripple-carry.

MULTIPLEXER



Dal punto di vista della definizione in VHDL, bisogna definire la sua **entity** (ovvero l'interfaccia che identifica ingressi e uscite del circuito) e la sua **architecture** (ovvero il suo comportamento e i suoi calcoli).

La funzione che descrive un mux è la seguente:

$$Y = (A \text{ AND } (\text{NOT } S)) \text{ OR } (B \text{ AND } S)$$

A destra è possibile infatti visionare la tabella di verità:

Un **multiplexer** (detto anche mux) è un elemento circuitale in grado di ricevere 2^n segnali in input insieme a n segnali di controllo. Quest'ultimi si occupano di individuare quale degli ingressi deve essere selezionato e incanalato verso l'unica uscita di cui dispone il circuito.

Vista la sua funzione è chiamato anche **selettore**.

Input			Output
A	B	S	Y
1	1	0	1
1	0	0	1
0	1	0	0
0	0	0	0
1	1	1	1
1	0	1	0
0	1	1	1
0	0	1	0

SPECIFICHE MULTIPLEXER

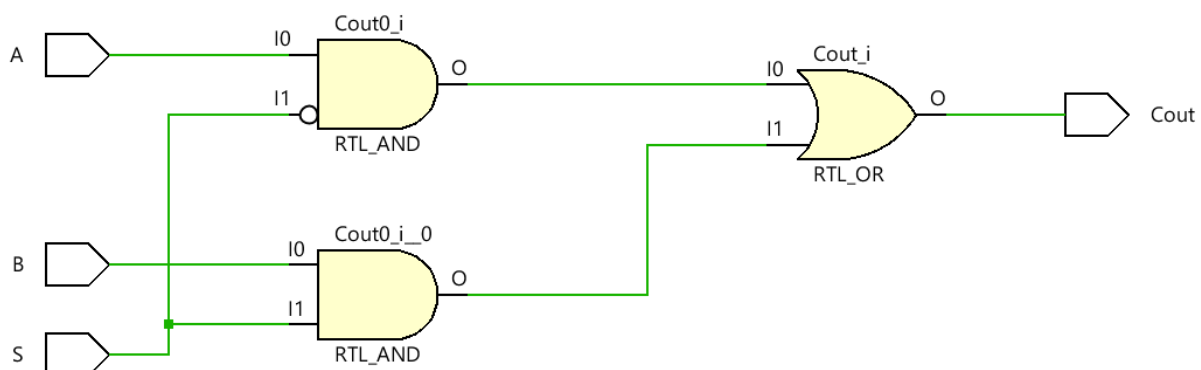
I dati utilizzati sono del tipo **std_logic**. Tale tipo fa parte della libreria IEEE e permette di risolvere problemi di conflitto di segnali in quanto rende possibile esprimere dei valori che vanno oltre ai semplici bit 0 e 1.

ENTITY E ARCHITECTURE

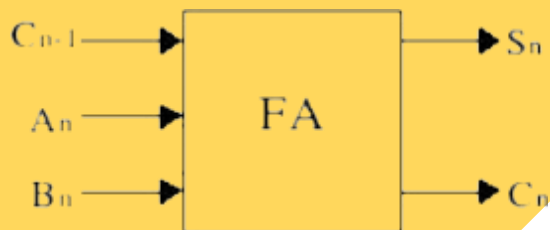
```
entity mux is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           S : in STD_LOGIC;
           Cout : out STD_LOGIC);
end mux;

architecture Behavioral of mux is
begin
    Cout <= (A and (not S)) or (B and S);
end Behavioral;
```

Cliccando su **Open Elaborated Design** nel menu RTL Analysis del progetto realizzato con Vivado, è possibile visualizzare l'implementazione a livello strutturale (detta anche gate-level)



FULL ADDER



Dal punto di vista della definizione in VHDL, dichiariamo la sua entity e la sua architecture.

Le funzioni che descrivono un full-adder sono le seguenti:

- Somma:

$$S = (A \text{ XOR } B) \text{ XOR } C_{IN}$$

- Riporto:

$$C_{OUT} = (A \text{ XOR } B) \text{ AND } C_{IN} \text{ OR } (A \text{ AND } B)$$

A destra è possibile infatti visionare la tabella di verità:

Un **full-adder** è un circuito sommatore che permette la somma di due bit ed è in grado di considerare anche un riporto.

Esso si compone a sua volta di due **half-adder**, i quali sono dei sommatore di due bit non in grado però di tenere in considerazione eventuale riporto in quanto caratterizzati da due soli ingressi. Ponendo, dunque, due half-adder in cascata in maniera opportuna è possibile sommare anche un eventuale riporto.

A_n	B_n	C_{n-1}	S_n	C_n
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

Tabella della verità del full adder

SPECIFICHE FULL ADDER

In questo script è possibile evidenziare due punti del linguaggio VHDL: i segnali e la dichiarazione delle porte.

I **signal** vengono utilizzati per rappresentare quei collegamenti nel circuito che non sono dunque riconducibili ad ingressi o uscite ma trasportano un segnale da una porta all'altra.

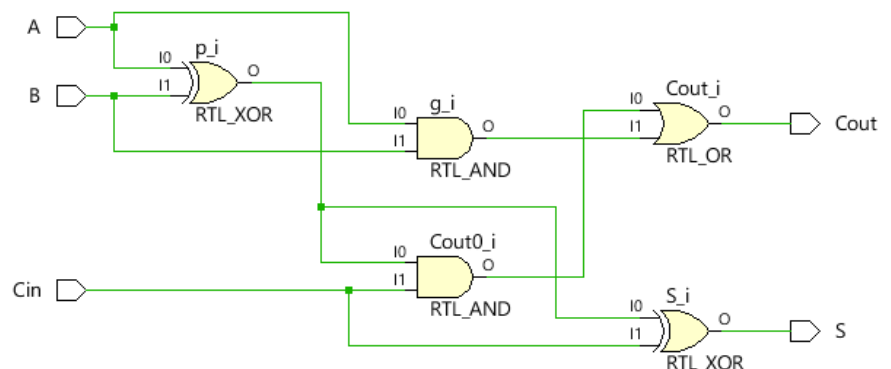
La **dichiarazione delle porte** può essere invece effettuata per ciascuna di esse o attraverso una dichiarazione cumulativa sulla stessa riga (a patto che le porte siano dello stesso tipo e con stessa direzione).

ENTITY E ARCHITECTURE

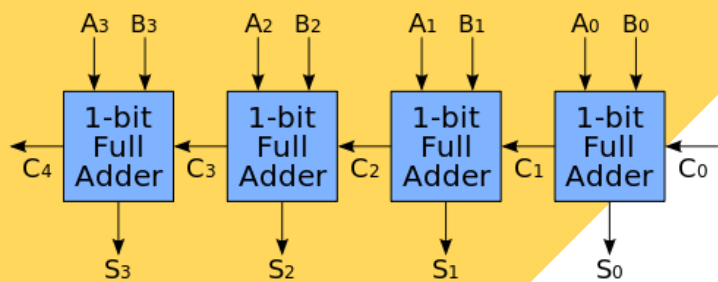
```
entity full_adder is
    Port ( Cin, A, B : in STD_LOGIC;
          S, Cout : out STD_LOGIC);
end full_adder;

architecture Behavioral of full_adder is
    signal p, g : STD_LOGIC;
begin
    S <= (A xor B) xor Cin;
    p <= A xor B;
    g <= A and B;
    Cout <= (p and Cin) or g;
end Behavioral;
```

Implementazione a livello strutturale:



RIPPLE CARRY



Come già visto nell'esempio del carry-select a 4 bit, un ripple-carry si compone di una serie di full-adder in cascata, collegati in modo che il riporto di ciascun componente sia il riporto in ingresso del successivo. Tale circuito è però caratterizzato da un ritardo intrinseco in quanto ogni full-adder, per poter svolgere il proprio calcolo correttamente, deve attendere il riporto ottenuto dal componente precedente.

Il ritardo di un ripple-carry, dunque, dipende dal numero di porte che un segnale attraversa.

Infatti, il **ritardo totale di propagazione** di un circuito è pari, approssimativamente, al ritardo della generica porta moltiplicato per i livelli logici. A riguardo si individua il **critical path** (percorso critico) ovvero il percorso col maggiore numero di porte nelle quale passa un segnale che, nel caso del full-adder, è pari a 3.

Tuttavia, possiamo assumere nel caso del ripple-carry che, tranne per il primo FA, i restanti avranno un path critico costituito da due porte in quanto il calcolo della somma $A+B$ viene eseguito subito senza attesa. Di conseguenza, per un ripple-carry a 4 bit il risultato sarà ottenuto dopo aver attraversato $3 + 2 \cdot 4 = 11$ porte; possiamo dunque definire che il ritardo sarà pari a **11 gate delays**.

SPECIFICHE RIPPLE CARRY

DETTAGLI IMPLEMENTATIVI

Come accennato precedentemente, le specifiche dei blocchi di un carry-select possono essere diverse: è possibile, infatti, costruire blocchi di uguale dimensione oppure di dimensioni differenti, ad esempio di grandezza crescente di una unità.

La tipologia trattata in questa relazione è quella di un simmetrico carry-select composto da tre ripple-carry a 8 bit ciascuno, permettendo così una buona parallelizzazione del calcolo.

I ripple-carry necessari sono tre poiché il primo blocco sarà costituito da un solo ripple-carry in quanto possiamo assumere che il resto in ingresso sia pari a zero. Invece, per il restante blocco avremo due ripple-carry.

In particolare, è necessario dunque ricorrere a delle strutture dati quali i **vector** che permettono la gestione della dimensione dei dati.

Inoltre, il codice è **parametrico** ovvero fa uso di un parametro definito tramite il comando `generic` che permette la riusabilità e la chiarezza del codice. Tuttavia, nel processo di sintesi (ovvero di realizzazione del circuito) tale valore deve essere fissato ad un intero.

In particolare, andrò ad utilizzare un ripple-carry parametrico che fa uso anche di due altri importanti comandi quali il **for generate**, che serve ad automatizzare l'istanziamento di componenti uguali, e l'istruzione **port map** che viene utilizzata per istanziare un componente mappando i segnali alle porte del circuito.

SPECIFICHE RIPPLE CARRY

Nel codice si ricorre ad un array di riporti con dimensione pari a $n+1$ in modo da tenere in considerazione un eventuale presenza di **overflow** che si verifica quando il numero di bit degli operandi non è sufficiente a rappresentare il risultato che richiede, appunto, un bit in più. Inoltre, il codice risulta essere parametrico.

Come già visto, si ricorrerà a 3 ripple-carry da 8 bit che posso ottenere semplicemente modificando il parametro generic nel seguente modo:

```
generic (n: integer := 8);
```

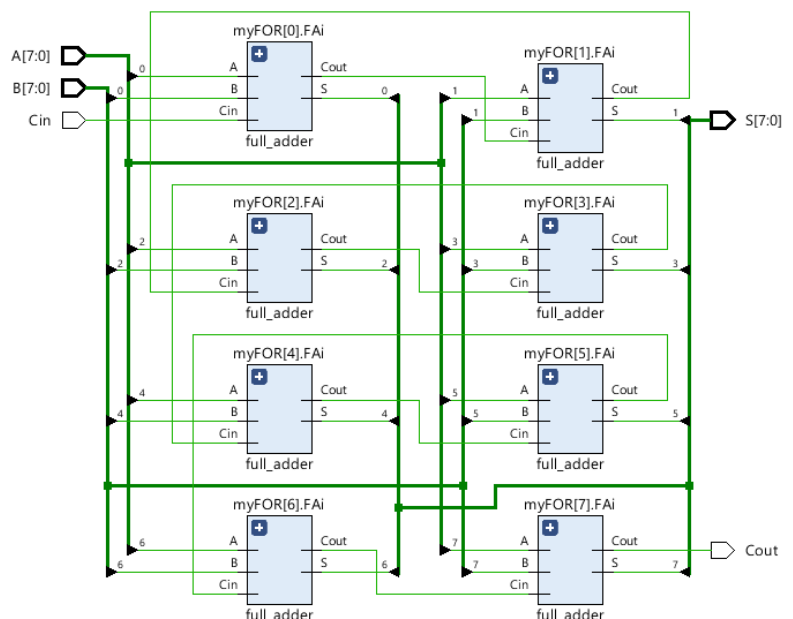
ENTITY E ARCHITECTURE

```
entity RCA is
    generic (n: integer);
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
          B : in STD_LOGIC_VECTOR (n-1 downto 0);
          Cin : in STD_LOGIC;
          Cout : out STD_LOGIC;
          S : out STD_LOGIC_VECTOR (n-1 downto 0));
end RCA;

architecture Behavioral of RCA is
    component full_adder is
        port(A, B, Cin: in STD_LOGIC;
             Cout, S: out STD_LOGIC);
    end component;
    signal C : STD_LOGIC_VECTOR (n downto 0);

begin
    myFOR: for i in 0 to n-1 generate
        FAi: full_adder port map (A(i), B(i), C(i), C(i+1), S(i));
    end generate;
    C(0) <= Cin;
    Cout <= C(n);
end Behavioral;
```

Una volta definito il parametro dei blocchi, è possibile visionare l'elaborated design che evidenzia la presenza di otto full-adder opportunamente collegati.



SPECIFICHE CARRY SELECT

```
entity carry_select is
  generic(n: integer);
  port( Acs : in STD_LOGIC_VECTOR (n-1 downto 0);
        Bcs : in STD_LOGIC_VECTOR (n-1 downto 0);
        Scs : out STD_LOGIC_VECTOR (n downto 0));
end carry_select;

architecture Behavioral of carry_select is
  -- definisco il full-adder
  component full_adder is
    port(A, B, Cin: in STD_LOGIC;
          Cout, S: out STD_LOGIC);
  end component;

  -- definisco il ripple_carry con una dimensione pari alla
  -- metà di quella del carry-select
  component RCA is
    port ( A : in STD_LOGIC_VECTOR ((n/2)-1 downto 0);
           B : in STD_LOGIC_VECTOR ((n/2)-1 downto 0);
           Cin : in STD_LOGIC;
           Cout : out STD_LOGIC;
           S : out STD_LOGIC_VECTOR ((n/2)-1 downto 0));
  end component;

  -- definisco il multiplexer
  component mux is
    port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           S : in STD_LOGIC;
           Cout : out STD_LOGIC);
  end component;

  -- definisco dei segnali intermedi per i riporti ottenuti
  signal Cint, Cout0, Cout1, Cout, Cof: STD_LOGIC;
  -- definisco due vettori per salvare la somma ottenuta dai
  -- due ripple carry
  signal Sout0, Sout1: STD_LOGIC_VECTOR ((n/2)-1 downto 0);
```

```
begin
  -- definisco il primo blocco che elabora gli 8 bit meno significativi
  RCA0: RCA port map(Acs((n/2)-1 downto 0), Bcs((n/2)-1 downto 0), '0', Cint, Scs((n/2)-1 downto 0));
  RCA1: RCA port map(Acs(n-1 downto (n/2)), Bcs(n-1 downto (n/2)), '0', Cout0, Sout0);
  RCA2: RCA port map(Acs(n-1 downto (n/2)), Bcs(n-1 downto (n/2)), '1', Cout1, Sout1);

  -- definisco i multiplexer che calcolano il bit risultante da ciascuna somma
  myFor: for i in 0 to ((n/2)-1) generate
    MUXi: mux port map(Sout0(i), Sout1(i), Cint, Scs((n/2)+i));
  end generate myFor;
  -- attraverso un altro multiplexer valuto quale dei due risultati calcolati è corretto
  MUXf: mux port map(Cout0, Cout1, Cint, Cout);
  -- gestisco un eventuale overflow tramite un full-adder posto alla fine
  Faf: full_adder port map(Acs(n-1), Bcs(n-1), Cout, Cof, Scs(n));
end Behavioral;
```

Nel ripple-carry, come già visto, ogni coppia di bit viene sommata due volte (da due full-adder diversi, tenendo in considerazione i due possibili riporti) e i due risultati sono memorizzati. Ad ognuna di queste coppie è associato poi un multiplexer che valuta quale dei due risultati sia corretto in base al riporto ottenuto dal full-adder precedente.

In questo caso, dunque, 8 mux sono addetti a questo compito in quanto, come già detto, il calcolo è ridondante solo per gli 8 bit più significativi. Avremo, infatti, 4 full-adder e 4 multiplexer per ogni ripple-carry del secondo blocco.

Un ulteriore multiplexer serve a gestire la scelta fra i risultati dei due ripple-carry in base al riporto ottenuto dal primo blocco. Invece, per gestire opportunamente un eventuale overflow, si introduce un full-adder finale che ricalcola la somma dei due bit più significativi e riesce a gestire dunque anche gli operandi con segno.

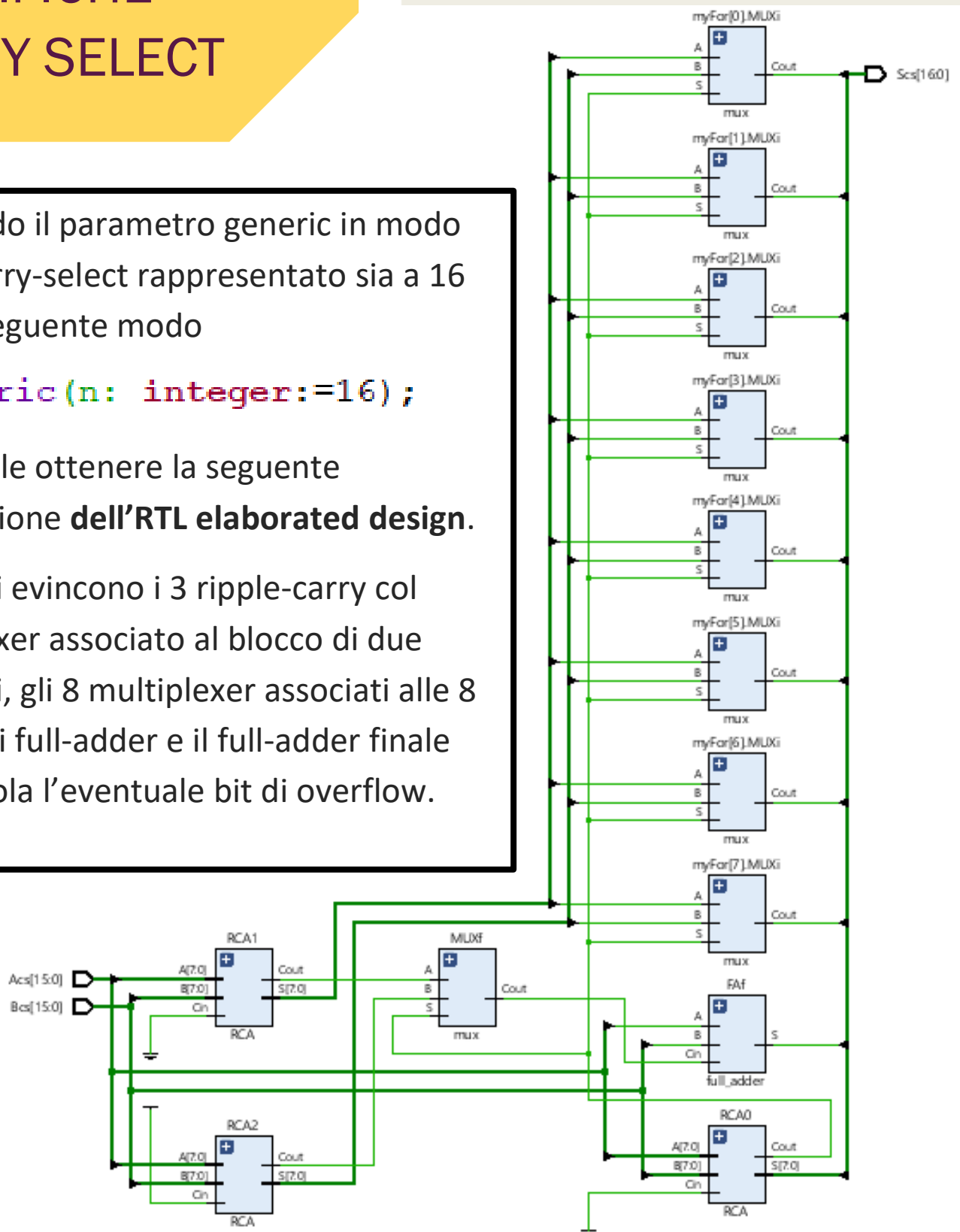
SPECIFICHE CARRY SELECT

Definendo il parametro generic in modo che il carry-select rappresentato sia a 16 bit nel seguente modo

```
generic(n: integer:=16);
```

è possibile ottenere la seguente
realizzazione **dell'RTL elaborated design.**

In essa si evincono i 3 ripple-carry col multiplexer associato al blocco di due elementi, gli 8 multiplexer associati alle 8 coppie di full-adder e il full-adder finale che calcola l'eventuale bit di overflow.



TESTBENCH

Un **testbench** 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.

```
entity sim_1 is
end sim_1;
-- vuoto in quanto è una simulazione
architecture Behavioral of sim_1 is
    component carry_select is
        port( Acs : in STD_LOGIC_VECTOR (15 downto 0);
              Bcs : in STD_LOGIC_VECTOR (15 downto 0);
              Scs : out STD_LOGIC_VECTOR (16 downto 0));
    end component;




    signal IA, IB : STD_LOGIC_VECTOR (15 downto 0);
    signal Sum : STD_LOGIC_VECTOR (16 downto 0);
begin
    IA <= "1011101011101010";
    IB <= "1001010101010111";

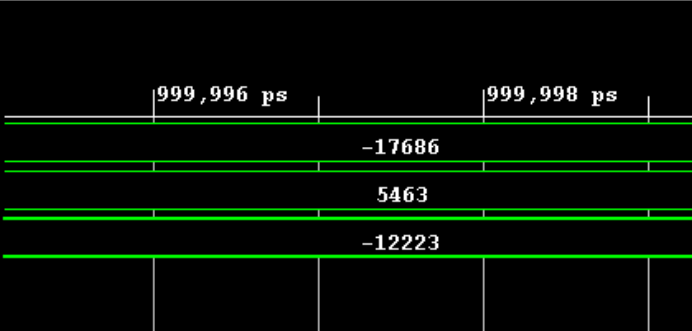
    CSA : carry_select port map(IA, IB, Sum);
end Behavioral;
```

In un testbench non sono presenti elementi di input o output ma vengono utilizzati dei segnali che conterranno i valori della simulazione.

Nel mio caso ho prima effettuato una simulazione che tenesse in considerazione una sola coppia di numeri binari quali 1011101011101010 e 0001010101010111 corrispondenti rispettivamente ai numeri -17686 e 5463 in notazione con segno.

Come già visto, la gestione del segno è affidata ad un full-adder.

Name	Value
>  IA[15:0]	-17686
>  IB[15:0]	5463
>  Sum[16:0]	-12223



TESTBENCH – SIMULAZIONE ESAUSTIVA

Tuttavia, è ottimale effettuare una cosiddetta una simulazione esaustiva ovvero un test in cui si inviano in input tutte le possibili combinazioni di operandi; nel caso di numeri a 16 bit avremo dunque che il range di valori è $[-2^{n-1}, 2^{n-1}-1]$ ovvero $[-32768, 32767]$.

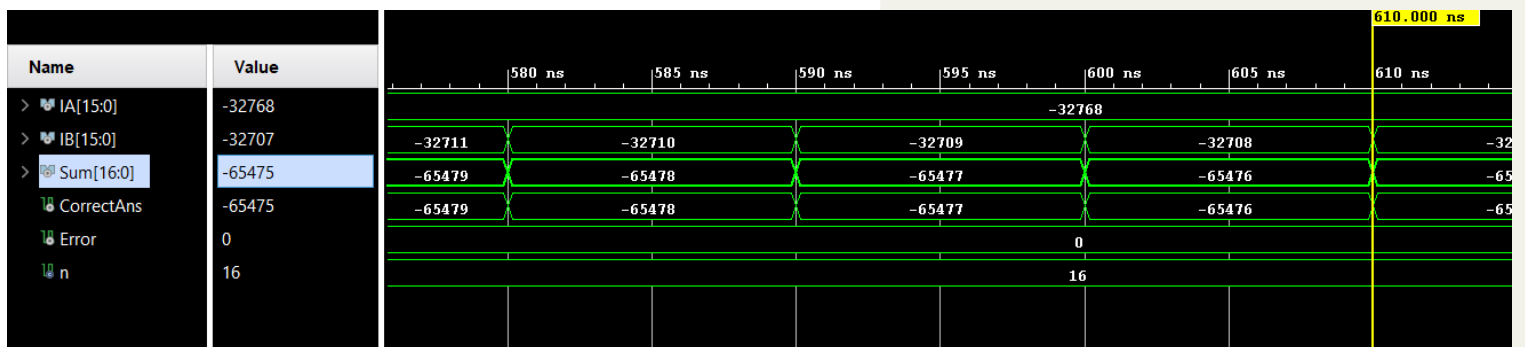
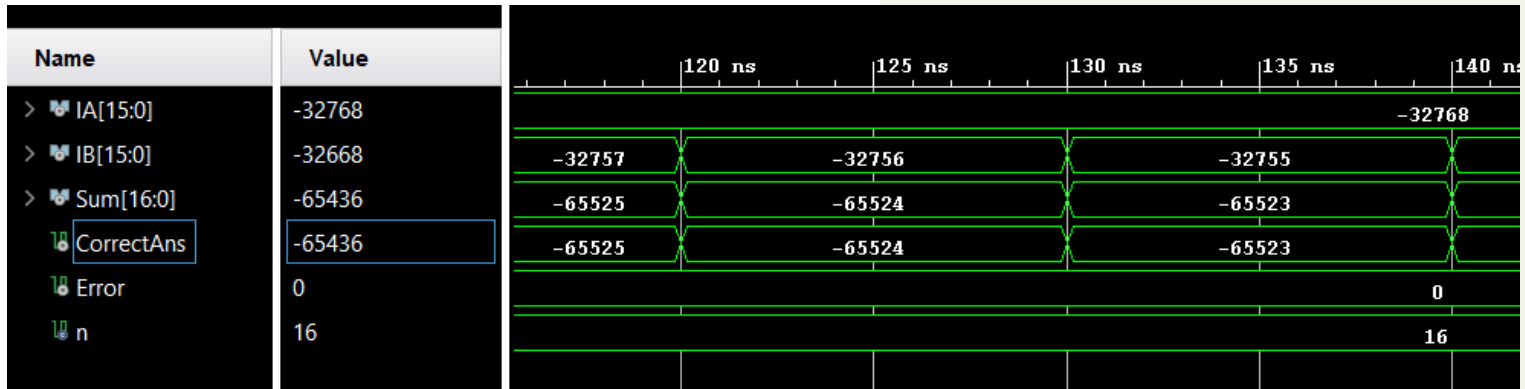
Per generare le varie combinazioni si può fare uso del costrutto **for loop** che assegna alle variabili di input un valore per volta.

In particolare, i valori corrisponderanno agli indici del ciclo di for sono degli interi e dunque, per poter essere assegnati, devono essere convertiti attraverso la funzione **conv_std_logic_vector** che posso utilizzare importando la libreria STD_LOGIC_arith.

Introduco, inoltre, una variabile di errore Error che calcola la differenza fra il risultato Sum ottenuto dal circuito e una somma algebrica CorrectAns calcolata tramite l'operatore di addizione.

Inoltre, utilizzo in questo caso una variabile generic che, nel momento del lancio della simulazione, inizializzo al valore 16.

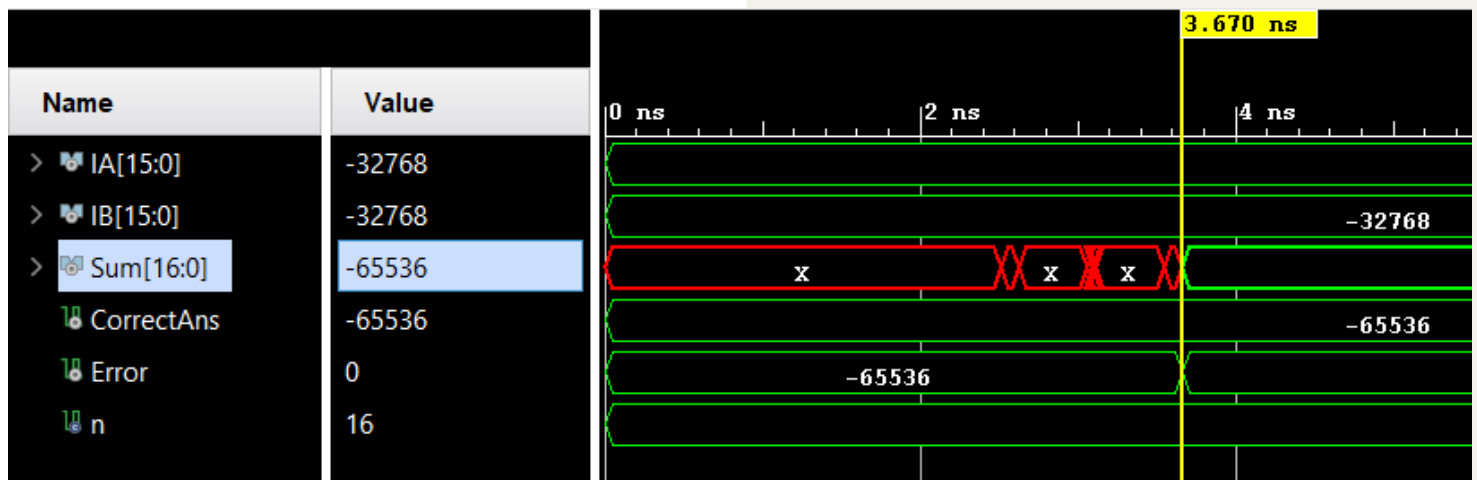
BEHAVIORAL SIMULATION



Facendo partire la simulazione è dunque evidente come l'errore calcolato sia sempre zero in quanto lo scarto fra il calcolo corretto e quello effettuato dal circuito è nullo. In particolare, sono qui mostrati due intervalli di simulazione.

Tuttavia, la Behavioral simulation non tiene in considerazione delle specifiche reali di un circuito. Posso allora eseguire una simulazione detta **post-synthesis timing simulation** che mi permette di effettuare delle osservazioni sui tempi di risposta.

POST SYNTHESIS TIMING SIMULATION

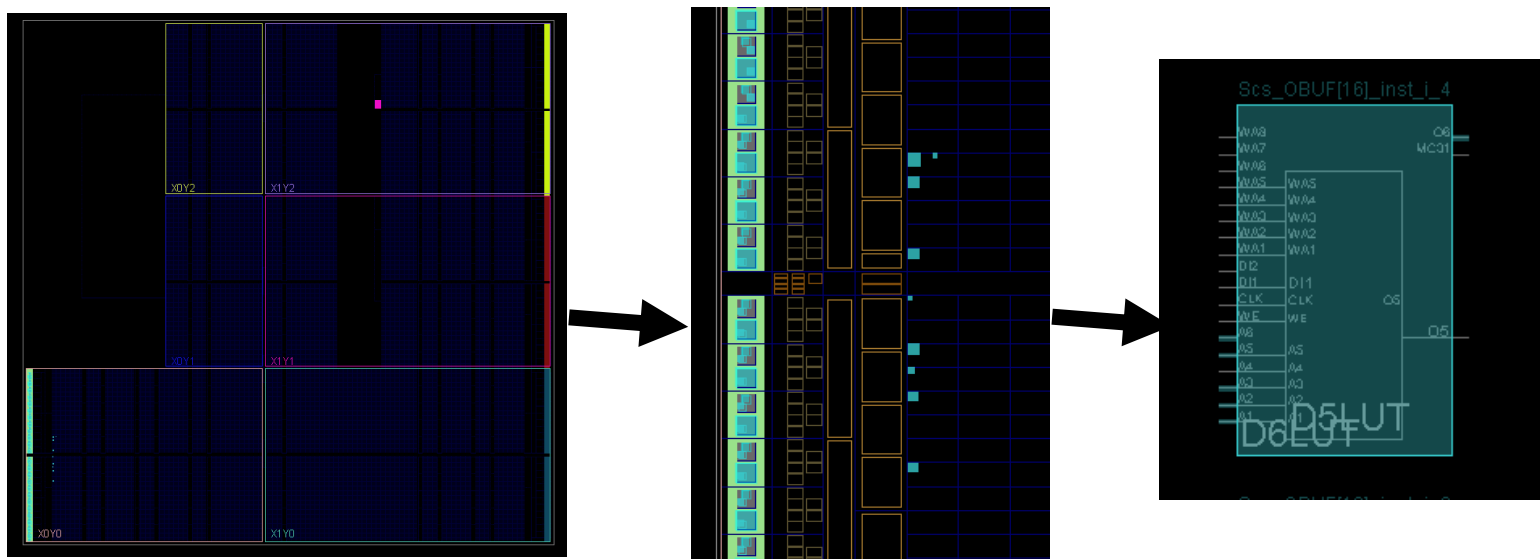


In particolare, attraverso la Post-Synthesis Timing Simulation è evidente la presenza di una non specificazione 'X' la quale indica che vi è una elaborazione degli ingressi in corso, evidenziando dunque che vi è un ritardo dovuto ai componenti.

Il ritardo iniziale presente sarà pari a 3.670 ns ed è inoltre evidente come, con cadenza periodica, il valore di errore è pari a 1 a causa appunto della presenza di ritardo.

In aggiunta, è possibile una analisi ancora più approfondita attraverso la realizzazione dell'implementazione del circuito.

POST IMPLEMENTATION SIMULATION



Operando la **Post-Implementation Simulation** è possibile visualizzare quali componenti del device a nostra disposizione sono utilizzati dal circuito; essi sono colorati in blu.

In particolare, il nostro device fa uso di **look up table** ovvero di strutture dati che associano determinati dati in ingresso a degli output memorizzati appunto sottoforma di tabelle. Il meccanismo delle LUT è molto utile in quanto è molto più efficiente accedere ad un'area di memoria piuttosto che operare da capo calcoli molto complessi.

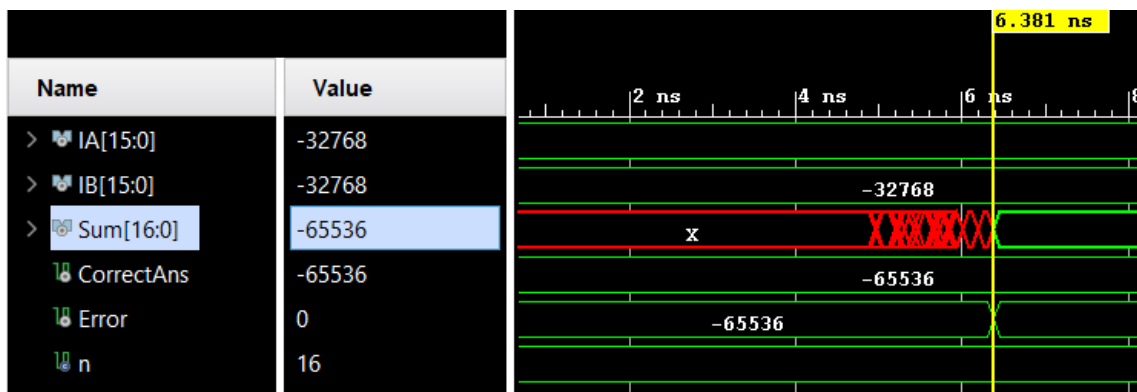
POST IMPLEMENTATION TIMING SIMULATION

Inoltre, attraverso la sezione **Truth table** è possibile visualizzare le tabelle di verità che definiscono le funzioni di ciascuna LUT.

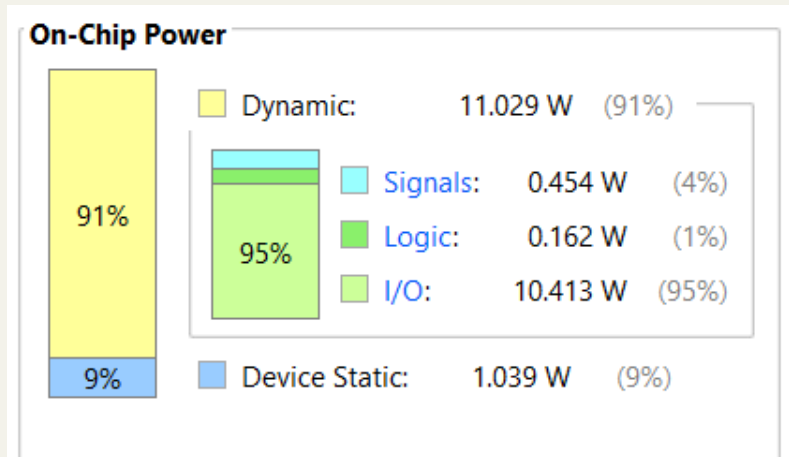
Scs_OBUF16_inst_4

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 + I1 & I2 & I3 & I4
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	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	1

Operando invece una **Post-Implementation Timing Simulation** è evidente che il ritardo in questo caso è pari a 6.381 ns, ovvero maggiore di quello ottenuto tramite la post-synthesis simulation a causa dell'implementazione su device. Infatti, in questo caso il comportamento rappresentato è proprio quello di un carry-select presente su un chip reale. Inoltre, anche in questo caso è ancora presente una periodicità nell'errore di calcolo, sempre a causa del ritardo.



REPORT



Operate le simulazioni, si ottengono anche dei **file di report** ovvero delle descrizioni dettagliate riguardo consumi ed efficienza del circuito.

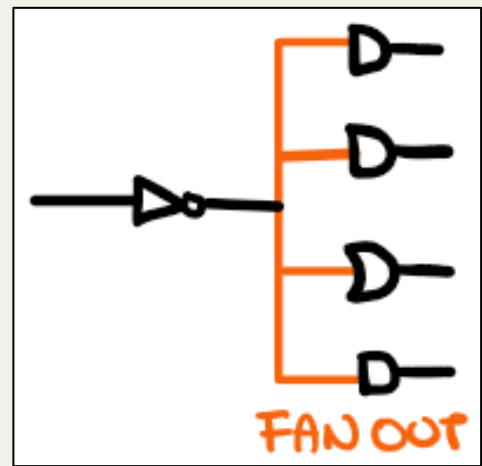
Nel mio caso, nel grafico On-Chip Power è evidente un consumo statico pari al 9% e un consumo dinamico pari al 91%.

In particolare, col termine **potenza statica dissipata** si intende una condizione in cui l'uscita del circuito è ad un valore costante (alto o basso che sia); con **potenza dinamica dissipata** invece si intende la dissipazione che avviene nella commutazione dell'uscita da un livello all'altro.

Il consumo di dynamic power è pari a $P = C V^2 F$ dove P si misura in Watt ed è il consumo dinamico, V è la tensione di funzionamento in Volt, F è la frequenza misurata in kHz e C è la capacità di carico in μF .

Inoltre, è importante definire il **fan-out** di una porta in quanto l'uscita di una qualsiasi porta può fornire un limitato valore di corrente per funzionare in modo corretto; se, invece, essa deve fornire una corrente maggiore di tale valore limite, allora si dice che è sovraccaricata (**overloaded**) e cessa di funzionare nel modo corretto.

CIRCUITI - APPROFONDIMENTO



Per **fan-out** di una porta logica si intende dunque il numero di porte logiche che possono essere collegate alla sua uscita garantendo il trasferimento dell'informazione. In pratica esiste un massimo numero superato il quale il funzionamento del circuito non è più garantito a causa dell'impossibilità di fornire la corrente necessaria a tutte le porte successive, con possibilità dunque che il risultato venga alterato. I massimi limiti in genere stabiliti dalla data famiglia logica o dai datasheet del produttore del dispositivo.

Inoltre, l'implementazione delle porte logiche nella realtà avviene tramite la combinazione di **transistor**. Per poter ottenere una sempre maggiore miniaturizzazione dei sistemi informatici su singolo chip, si tende a diminuire sempre più le dimensioni dei dispositivi elementari per poterne aumentare il numero, fenomeno detto **scaling**.

Questo segue la nota **legge di Moore**, la quale afferma che ogni due anni il numero di transistor di un processore raddoppia.

CIRCUITI - APPROFONDIMENTO

Legge di Moore

Microprocessor transistor counts 1971-2011 & Moore's law



La riduzione delle dimensioni dei transistor, tuttavia, è sottoposta a vincoli di varia natura:

- **progettuale**, a causa ad esempio dell'aumento della complessità delle interconnessioni;
- **termica**, poiché un aumento del numero di transistor operanti nello stesso chip porta ad un aumento anche del calore dissipato e della temperatura del chip;
- **robustezza**, dato che, ad esempio, aumentano gli accoppiamenti fra le varie componenti del circuito e quindi aumentano i disturbi.

Inoltre, un aumento del numero di transistor in un unico dispositivo ha come principale effetto negativo l'aumento del consumo di potenza, che risulta dannoso soprattutto nei sistemi portatili alimentati a batterie.