



Progetto di Elettronica Digitale

Giorgio Ubbriaco
209899

Indice

INTRODUZIONE	3
MYDEF.....	4
REG.....	4
<i>Clock</i>	<i>4</i>
<i>Clear.....</i>	<i>4</i>
DESCRIZIONE REGISTRO	5
RTL ANALYSIS – SCHEMATIC.....	7
ADDER.....	8
DESCRIZIONE SOMMATORE	8
RTL ANALYSIS – SCHEMATIC.....	9
PIPELINECIRCUIT4OPERANDS	10
DESCRIZIONE.....	10
PC40	12
RTL ANALYSIS – SCHEMATIC.....	15
PIPELINECIRCUIT4OPERANDS SIMULATION	16
BEHAVIORAL SIMULATION	19
SYNTHESIS.....	20
<i>Synthesis – Schematic</i>	<i>20</i>
<i>Post-Synthesis Timing Simulation.....</i>	<i>21</i>
<i>Post-Synthesis Report.....</i>	<i>22</i>
<i>Constraint</i>	<i>23</i>
IMPLEMENTATION	25
<i>Implementation – Device</i>	<i>25</i>
<i>Post-Implementation Timing Simulation</i>	<i>26</i>
<i>Post-Implementation Report</i>	<i>27</i>
LATENZA	28
THROUGHPUT	28
MASSIMA FREQUENZA DI FUNZIONAMENTO.....	28
POWER ON-CHIP	32

Introduzione

Progettare un circuito che, ricevuti in ingresso i **quattro operandi A, B, C e D ad n-bit in complemento a due**, ed il **segnale di controllo Contr a 2-bit**, svolga le seguenti operazioni:

Operazione
A+B+C+D se Contr="00"
A-B+C-D se Contr="01"
A+B-C+D se Contr="10"
A-B-C-D se Contr="11"

Il circuito deve sfruttare una **struttura pipeline** che garantisca una **latenza di due cicli di clock ed un throughput unitario**. Tutti i registri devono essere forniti di **clear asincrono** ed essere sensibili al **fronte di discesa del clock**. Si richiede di fornire la descrizione VHDL parametrica e di caratterizzare il circuito progettato in termini di: latenza, throughput, massima frequenza di funzionamento, occupazione di risorse e dissipazione di potenza, per **n=8 ed n=16**.

Nella relazione in formato pdf devono essere riportati:

1. I codici VHDL del circuito, del testbench e di eventuali files di support;
2. Una descrizione delle scelte effettuate;
3. Screenshot delle simulazioni eseguite (behavioral, post-synthesis e post-implementation);
4. Descrizione dei risultati ottenuti dai reports.

MyDef

Nel package MyDefinitions definisco la costante *n_bit* in riferimento al numero di bit che caratterizzerà il mio circuito e che mi permetterà di mantenere genericità nel codice VHDL con un semplice *import* del package stesso nei vari file di supporto al circuito.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 package MyDefinitions is
5     constant n_bit: integer:=16;
6 end package;
```

Figura 1 MyDef.vhd

Reg

Reg è il file VHDL contenente la descrizione di un registro generico sensibile ai fronti di discesa del Clock e fornito di Clear asincrono.

Clock

È un segnale periodico nel tempo. Esso va avanti indefinitamente nel tempo grazie al generatore di clock. Il segnale di clock è descritto sostanzialmente da due parametri:

1. Il periodo T è il tempo che intercorre tra due eventi;
2. La frequenza pari a $\frac{1}{T}$.

Clear

È un segnale che permette di forzare l'uscita di un Flip Flop a 0, cioè quando il segnale di clear sarà *alto* l'output Q verrà *pulito* dall'ingresso D e sarà posto a 0. Esso è un segnale *asincrono*, cioè il suo effetto non dipende dal clock. Inoltre, è anche prioritario, cioè fin quando è alto i fronti del clock verranno ignorati e, pertanto, se attivo sarà lui a decidere l'uscita di Q.

Descrizione Registro

Dato che ogni input ed ogni output generato dal circuito dovrà essere memorizzato e che ogni input/output è di n-bit generico, allora dovremo considerare moduli sequenziali che memorizzano ognuno un bit. Questi moduli sono conosciuti come Flip Flop i quali memorizzano, in base al fronte di salita o di discesa del clock, il bit dato in input ad esso. Inoltre, dovranno essere forniti ognuno di clear asincrono. Pertanto, essendo che ci serviranno n Flip Flop potrò considerare un modulo sequenziale denominato Registro. Tale modulo non è altro che un gruppo di Flip Flop sensibili tutti allo stesso fronte di salita o di discesa del clock e forniti tutti dello stesso clear asincrono.

Affichè tutto funzioni correttamente bisogna rispettare dei vincoli temporali. Un registro funziona correttamente se e solo se vengono rispettati i vincoli definiti in termini di tempo di setup e tempo di hold. Il tempo di setup **tsetup** rappresenta il tempo in cui preventivamente l'ingresso del Flip Flop deve diventare stabile, cioè quanto tempo prima l'ingresso D deve diventare stabile al fronte di clock che avverrà successivamente. Dopo che il fronte di clock è passato, D non potrà cambiare subito essendo che non commuta in tempo istantaneo. Pertanto, prima di far cambiare nuovamente D devo aspettare un certo tempo, cioè un tempo di hold. Il tempo di hold **thold** rappresenta il minimo tempo tra il fronte di clock e la successiva variazione dell'ingresso. Generalmente il tempo di setup e il tempo di hold sono nei datasheet.

Ovviamente, bisogna evitare situazioni in cui Q processa valori non nominali quali 0 V e Vdd (generici). Nel caso in cui si prendono valori di tensione diversi da quelli rappresentativi dello '0' e dell'"1" logico si può creare lo stato di indeterminazione 'X', cioè il rischio di prendere valori di tensione ancora non stabili. La stessa situazione accade nel caso in cui la successiva variazione ricade anch'essa negli intervalli di tempo di *tsetup* e *thold*, cioè possono generare un cattivo trasferimento dell'informazione perché si rischia di beccare, in corrispondenza del fronte di clock, dei valori di tensione che non rappresentano né lo '0' né l'"1" logico. Tale situazione prende il nome di **METASTABILITÀ**. Essa rappresenta il *terzo stato logico* e può creare malfunzionamenti irrisolvibili. Affichè non avvenga una situazione del genere si potrebbe pensare di *spostare* il fronte di clock in avanti così che la risposta del circuito diventi valida anche nel caso peggiore. Pertanto, potrei aumentare la distanza temporale fra i due fronti T di *tsetup*. In genere, ***tsetup* >> *thold***. Quindi, la garanzia di rispettare i vincoli temporali si ha con **$T = t_{max} + t_{setup}$** .

Inserisco qui di seguito il codice rappresentativo del registro ad n bit generico

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Reg is
5     generic (n: integer);
6     port(
7         D: in std_logic_vector(n-1 downto 0);
8         CLK: in std_logic;
9         CLR: in std_logic;
10        Q: out std_logic_vector(n-1 downto 0)
11    );
12 end Reg;
13
14 architecture Behavioral of Reg is
15     begin
16         process(CLK,CLR)
17             begin
18             if(CLR='1')then
19                 Q<=(others=>'0');
20             elsif(falling_edge(CLK))then
21                 Q<=D;
22             end if;
23         end process;
24 end Behavioral;
```

Figura 2 Reg.vhd

Il registro come già annunciato precedentemente presenterà l'ingresso D ad n-bit, il segnale di clock, il segnale di clear e l'uscita Q ad n-bit. Definiamo la sequenzialità tramite lo statement *process* indicando la *sensitivity-list*, cioè l'insieme dei segnali che *sanciranno* la sequenzialità del componente. Pertanto, se il segnale di clear sarà alto tutti i bit dell'uscita Q verranno posti a zero, mentre se siamo nel caso di un fronte di discesa del clock l'output Q memorizzerà il valore dell'input D. Ovviamente, il caso banale (*else*) è quello in cui Q mantiene il suo valore precedente ($Q \leq Q$).

RTL Analysis – Schematic

Allego qui di seguito lo *schematic* relativo al registro Reg ad n-bit generico:

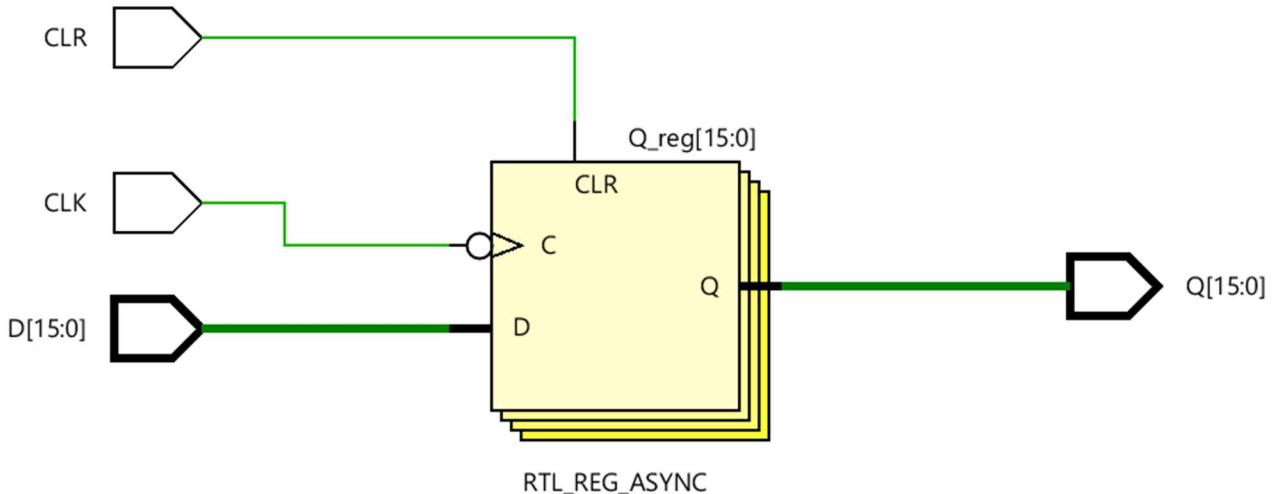


Figure 3 Reg.vhd Schematic

Possiamo notare l'ingresso D ad n-bit del registro, il segnale di clock, il segnale di clear ed, infine, l'uscita Q ad n-bit.

Adder

Descrizione Sommatore

Le operazioni che tale circuito può eseguire possono essere ricondotte tutte quante ad operazioni di somma. Infatti, le operazioni di sottrazione possono essere ricondotte a semplici operazioni di somma dove l'operando con segno negativo viene *trasformato* tramite complemento a due in modo che l'operazione da svolgere sarà sempre un'operazione di somma.

Il sommatore utilizzato è un *carry look ahead*. Tale sommatore migliora la velocità riducendo la quantità di tempo necessaria per determinare i bit di carry.

Determino i bit di somma e di carry utili per determinare il risultato del sommatore:

$$S_i = a_i \oplus b_i \oplus c_i$$
$$c_{i+1} = (a_i \oplus b_i) \cdot c_i + a_i \cdot b_i$$

Definisco i segnali di propagate e di generate:

$$p_i = a_i \oplus b_i$$
$$g_i = a_i \cdot b_i$$

Allego qui di seguito un esempio di Carry Look Ahead a 4-bit:

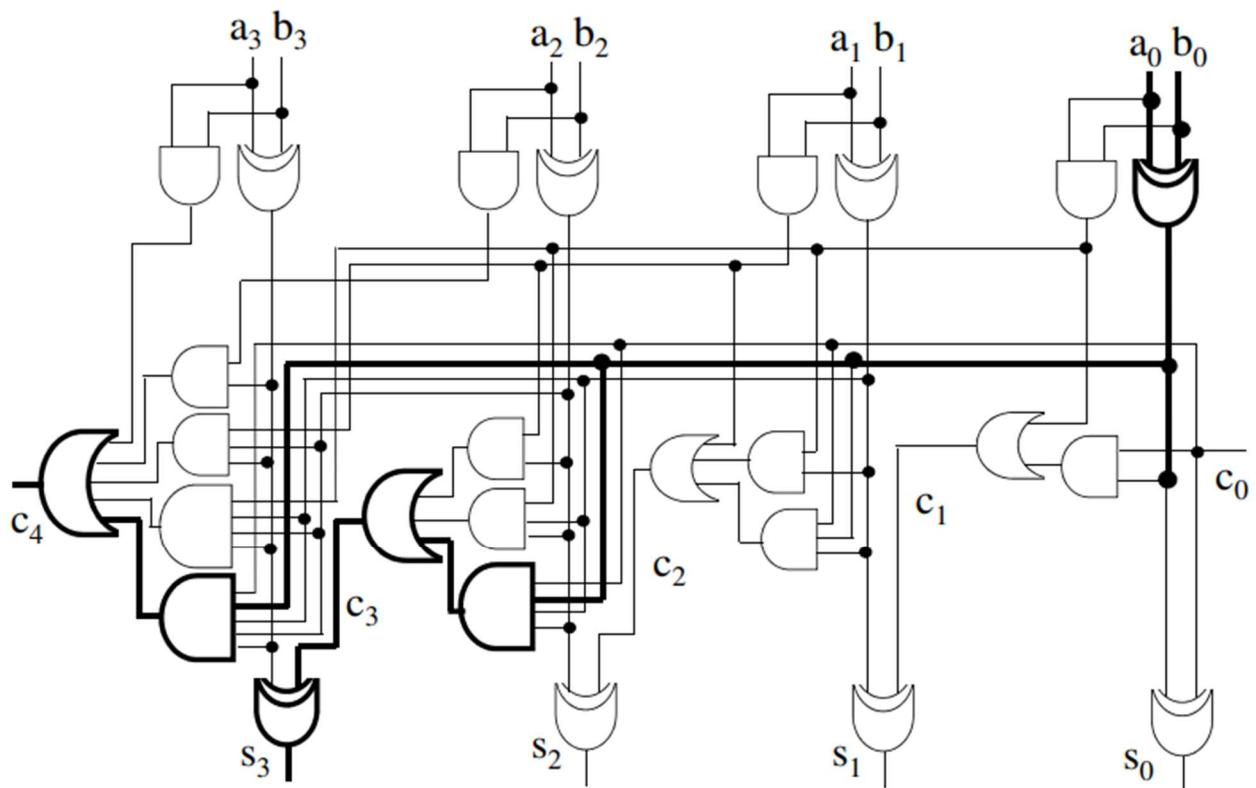


Figure 4 Carry Look Ahead a 4-bit

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Adder is
5     generic(n:integer);
6     port(
7         input1: in std_logic_vector(n-1 downto 0);
8         input2: in std_logic_vector(n-1 downto 0);
9         c0: in std_logic;
10        ris: out std_logic_vector(n downto 0)
11    );
12 end Adder;
13
14 architecture Behavioral of Adder is
15     signal p,g: std_logic_vector(n downto 0);
16     signal c: std_logic_vector(n+1 downto 0);
17
18 begin
19     c(0) <= c0;
20     p <= (input1(n-1) xor input2(n-1)) & (input1 xor input2);
21     g <= (input1(n-1) and input2(n-1)) & (input1 and input2);
22     ris <= p xor c(n downto 0);
23     c(n+1 downto 1) <= g or (p and c(n downto 0));
24
25 end Behavioral;

```

Figure 5 Adder.vhd

Il codice vhdl del componente Adder mantiene anch'esso la genericità sugli n-bit considerati. Esso è composto da due input, *input1* ed *input2*, da n-bit ciascuno, un riporto in entrata *c0* che ci permette di calcolare una somma o una differenza (nel secondo caso considerando ovviamente l'operando negativo con l'operazione not) e, infine, l'output *ris* ad n+1-bit. Nell'*architecture Behavioral* ho definito i segnali *propagate* e *generate* da n-bit ciascuno e il segnale carry da n+2-bit. Infine, nel corpo della begin ho calcolato i rispettivi segnali tramite le operazioni logiche già sopra citate.

RTL Analysis – Schematic

Allego qui di seguito lo schematic relativo all'Adder precedentemente descritto:

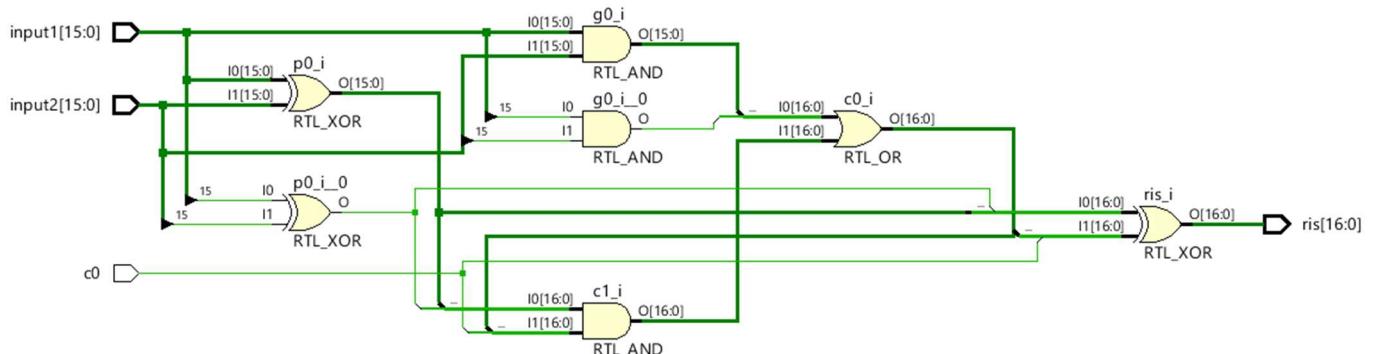


Figure 6 Adder.vhd Schematic

PipelineCircuit4Operands

Descrizione

Per realizzare il circuito previsto, utilizzo inizialmente 4 registri per memorizzare gli input del circuito stesso e 3 *multiplexer* così da stabilire quale tra l'uscita del registro diretta e negata (in riferimento a B, C e D) sarà il valore corretto da utilizzare. Successivamente, essendo che devo calcolare il complemento a due di C e questo causerebbe un aumento dei bit dell'operando C, uniformo il numero di bit anche per gli operandi A, B e D tramite una semplice somma per zero. Ovviamente, l'operazione di somma per C verrà sancita dal riporto in entrata a tale Adder: se tramite le condizioni imposte sul segnale di controllo *Contr* ottengo un *Cin* pari a zero significa allora che l'operando C è positivo mentre se ottengo un *Cin* pari a uno significa allora che l'operando C è negativo (tutto ciò gestito da un mux avente come selettore le condizioni imposte sul segnale di controllo *Contr*). Per quanto riguarda, invece, le operazioni di *somma per zero* di A, B e D, esse avranno come riporto in entrata zero. Infatti, il complemento a due per i possibili valori negativi di B e D verranno gestiti direttamente nell'operazione di somma generale e rispettivamente di AB e CD. Tanto è vero che le operazioni di somma AB e CD presenteranno come riporto in entrata un valore stabilito da un mux avente come selettore sempre le opportune condizioni sul segnale di controllo *Contr*. Quindi, le operazioni di somma AB e CD avranno come input signal ad $n+1$ bit e output signal ad $n+2$ bit. Ogni risultato ottenuto nelle operazioni di somma AB e CD verrà memorizzato nel registro corrispondente tale che le uscite di questi due registri ad $n+2$ bit ciascuna verrà mandata in input al sommatore finale ABCD avente come riporto in entrata lo zero essendo che ogni caso di complemento a due è stato preventivamente già gestito. L'uscita dell'Adder finale ABCD sarà ad $n+3$ bit ed essa verrà memorizzata nel registro corrispondente tale che l'uscita di tale registro sarà effettivamente l'output del circuito completo (Ho omesso nello schema i segnali di *clock*, *clear* e di controllo *Contr* per non appesantire il tutto).

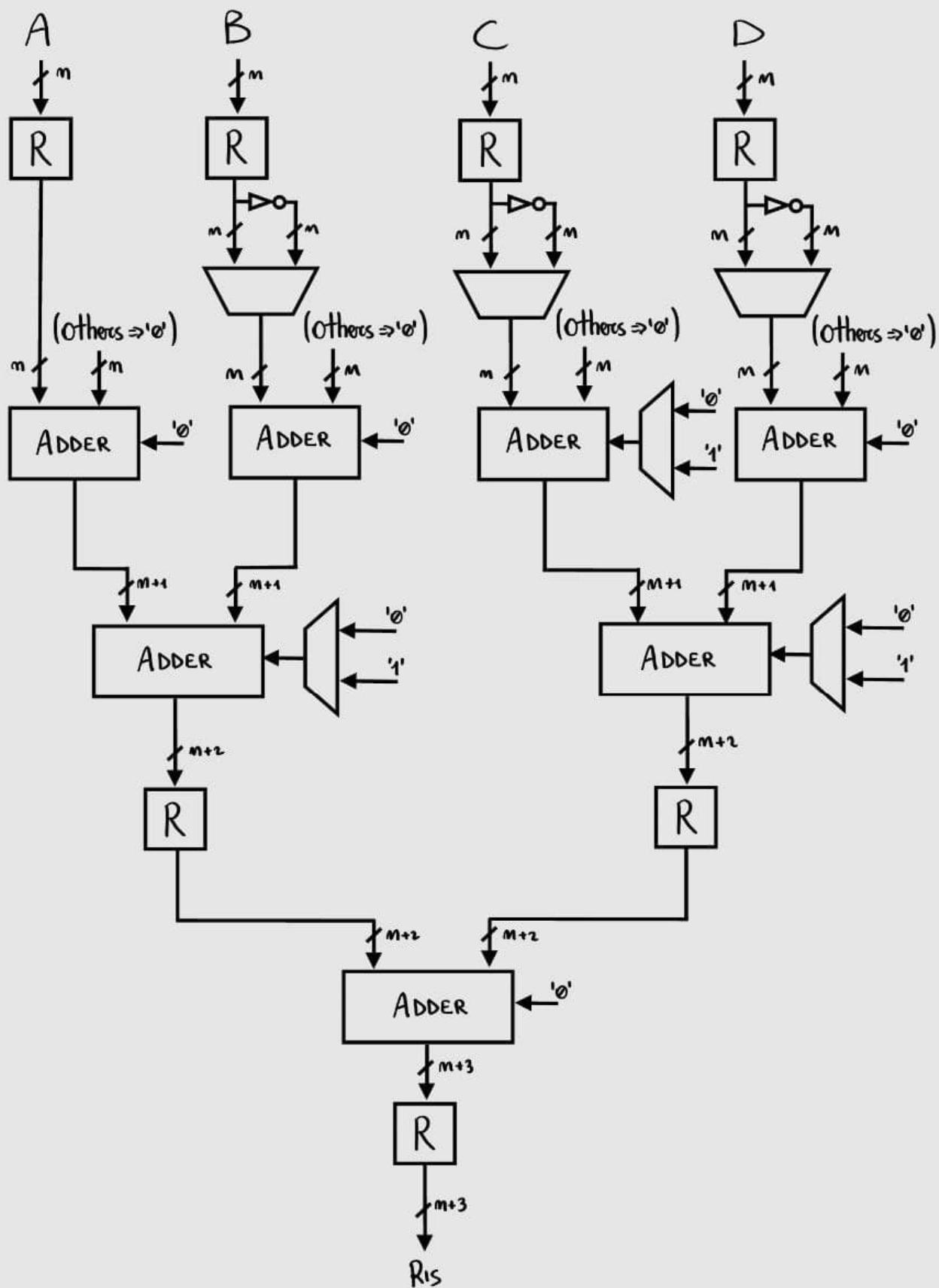


Figura 7 Schema del Pipeline Circuit 4 Operands

PC4O

Il seguente codice rappresenta il Pipeline Circuit 4 Operands:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 library work;
4 use work.MyDefinitions.all;
5
6 entity PC4O is
7     generic(n:integer:=n_bit);
8     port(
9         A,B,C,D: in std_logic_vector(n-1 downto 0);
10        Contr: in std_logic_vector(1 downto 0);
11        clk,clr: in std_logic;
12        Ris: out std_logic_vector(n+2 downto 0)
13    );
14 end PC4O;
15
16 architecture Behavioral of PC4O is
17 component Reg is
18     generic(n:integer);
19     port(
20         D: in std_logic_vector(n-1 downto 0);
21         CLK: in std_logic;
22         CLR: in std_logic;
23         Q: out std_logic_vector(n-1 downto 0)
24     );
25 end component;
26 component Adder is
27     generic(n:integer);
28     port(
29         input1: in std_logic_vector(n-1 downto 0);
30         input2: in std_logic_vector(n-1 downto 0);
31         c0: in std_logic;
32         ris: out std_logic_vector(n downto 0)
33     );
34 end component;
35
36 --signal corrispondenti ai valori negati di B,C,D
37 signal notB,notC,notD: std_logic_vector(n-1 downto 0);
38 -- signal corrispondenti agli input diretti e alle uscite di ogni registro
39 signal sA,sB,sC,sD: std_logic_vector(n-1 downto 0);
40 -- signal uscenti dai mux per decidere il valore diretto o negato
41 signal muxB,muxC,muxD: std_logic_vector(n-1 downto 0);
42 -- riporto in entrata per l'Adder C per calcolarne il positivo o il negativo
43 signal CinC: std_logic;
44 -- signal uscenti dagli Adder dopo aver uniformato i bit per ognuno
45 signal trueA,trueB,trueC,trueD: std_logic_vector(n downto 0);
46 -- riporti in entrata agli Adder AB, CD, ABCD
47 signal CinAB,CinCD,CinABCD: std_logic;
48 -- risultati degli Adder AB, CD e le uscite dei registri corrispondenti
49 signal AB,CD,sAB,sCD: std_logic_vector(n+1 downto 0);
50 -- risultato dell'Adder ABCD e L'uscita del registro corrispondente
51 signal ABCD,sABCD: std_logic_vector(n+2 downto 0);
```

```

53 begin
54     notB<=not B; notC<=not C; notD<=not D;
55     -- implemento i registri corrispondenti agli ingressi A,B,C,D
56     RegA: Reg generic map(n) port map(A,clk,clr,sA);
57     RegB: Reg generic map(n) port map(B,clk,clr,sB);
58     RegC: Reg generic map(n) port map(C,clk,clr,sC);
59     RegD: Reg generic map(n) port map(D,clk,clr,sD);
60
61     -- implemento i mux corrispondenti ai valori di B,notB,C,notC,D,notD
62     muxB <= sB when Contr(0)='0' else
63         not B when Contr(0)='1' else
64             (others=>'X');
65     muxC <= sC when Contr(1)='0' else
66         not C when Contr(1)='1' else
67             (others=>'X');
68     muxD <= sD when Contr(0)='0' else
69         not D when Contr(0)='1' else
70             (others=>'X');
71
72     -- calcolo il riporto in entrata per l'Adder di C per calcolarne il positivo o il negativo
73     CinC <= '0' when Contr(1)='0' else
74         '1' when Contr(1)='1' else
75         'X';
76
77     -- implemento gli adder per uniformare i bit
78     AdderA: Adder generic map(n) port map(sA,(others=>'0'),'0',trueA);
79     AdderB: Adder generic map(n) port map(muxB,(others=>'0'),'0',trueB);
80     AdderC: Adder generic map(n) port map(muxC,(others=>'0'),CinC,trueC);
81     AdderD: Adder generic map(n) port map(muxD,(others=>'0'),'0',trueD);
82
83     -- calcolo i riporti in entrata per gli Adder AB,CD
84     CinAB <= '0' when Contr(0)='0' else
85         '1' when Contr(0)='1' else
86         'X';
87     CinCD <= '0' when Contr(0)='0' else
88         '1' when Contr(0)='1' else
89         'X';
90
91     -- implemento gli Adder AB e CD
92     AdderAB: Adder generic map(n+1) port map(trueA,trueB,CinAB,AB);
93     AdderCD: Adder generic map(n+1) port map(trueC,trueD,CinCD,CD);
94
95     -- implemento i registri per i risultati degli Adder AB e CD
96     RegAB: Reg generic map(n+2) port map(AB,clk,clr,sAB);
97     RegCD: Reg generic map(n+2) port map(CD,clk,clr,sCD);
98
99     -- implemento l'Adder finale ABCD
100    AdderABCD: Adder generic map(n+2) port map(sAB,sCD,'0',ABCD);
101
102    -- implemento il registro corrispondente al risultato dell'Adder ABCD
103    RegABCD: Reg generic map(n+3) port map(ABCD,clk,clr,sABCD);
104
105    -- il risultato del circuito complessivo sarà:
106    Ris<=sABCD;
107
108 end Behavioral;

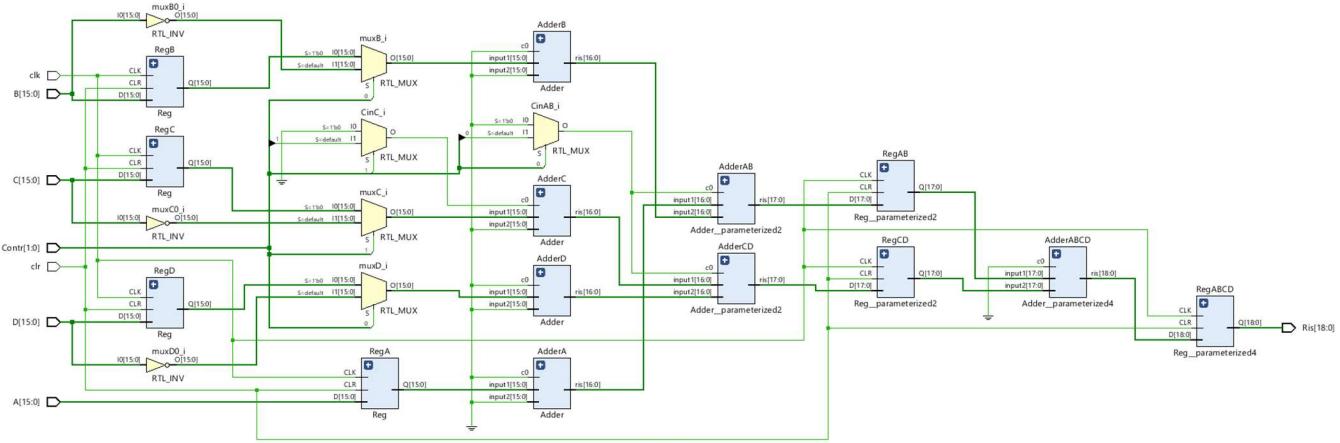
```

Il circuito in pipeline¹ a 4 operandi (*Pipeline_Circuit_4_Operands – PC4O*) presenta 6 ingressi tra cui i 4 operandi ad n-bit, il segnale di controllo *Contr* a 2-bit, i segnali di clock e clear e, infine, presenta un’uscita, cioè il risultato del circuito in pipeline a n+3-bit. Nell’Architecture del *PC4O* dichiaro i due componenti che mi serviranno per realizzare il circuito in questione, cioè il registro per salvare i valori in input e i risultati ottenuti e l’Adder che servirà a svolgere sia le operazioni di somma sia le operazioni di differenza. Tanto è vero che l’operazione di differenza potrà essere ottenuta effettuando il complemento a due dell’operando che dovrà essere negativo. Quindi, se, ad esempio, bisogna calcolare il risultato di A-B, questa operazione può essere svolta come A+(-B), cioè A+(not B)+’1’. Inoltre, vengono dichiarati vari signal: quelli corrispondenti alle uscite di ogni registro corrispondenti ad ogni input del circuito (*sA, sB, sC, sD*), i signal uscenti dai mux per decidere se considerare il valore diretto o negato nel caso in cui bisogna considerare i valori negati *-B, -C, -D*, il riporto in entrata per l’Adder C per calcolarne il corrispettivo valore positivo o negativo, i signal uscenti dagli Adder dopo aver uniformato i bit per ognuno ad n+1 bit (*trueA, trueB, trueC, trueD*), i riporti in entrata agli Adder *AB, CD* e *ABCD*, i risultati degli Adder *AB* e *CD* e i corrispondenti signal in uscita dai registri ad n+2 bit ciascuno e i signal per il risultato dell’Adder finale *ABCD* e l’uscita corrispondente del registro *sABCD* ad n+3 bit ciascuno. Nella *begin* dichiaro inizialmente i registri corrispondenti agli input del circuito ed implemento i mux per decidere quale tra il valore diretto o negato corrispondente dovrà essere considerato (ovviamente tenendo conto delle condizioni imposte sul segnale di controllo *Contr*). Successivamente calcolo il riporto in entrata per la somma dell’operando C per calcolarne caso mai il valore negativo. Dopodiché implemento gli Adder corrispondenti ad *A, B, C* e *D* che serviranno per uniformare l’aumento dei bit a causa del calcolo del possibile valore negativo di *C*. Calcolo, poi, i riporti in entrata ai sommatori *AB* e *CD*. Implemento tali sommatori e salvo i loro risultati nei corrispondenti registri. Infine, implemento l’Adder *ABCD* tenendo conto di un riporto in entrata pari a zero poiché tutte le operazioni sui numeri negativi sono state già effettuate. Come ultima operazione viene assegnata l’uscita del registro corrispondente al risultato dell’Adder finale *ABCD* all’output del circuito generale.

¹ Un circuito in pipeline è una tecnica impiegata in elettronica per realizzare architetture di circuiti digitali ad alte prestazioni, che consiste nello scomporre una complessa elaborazione in una sequenza di operazioni più semplici, ognuna delle quali è effettuata da un singolo circuito di una cascata di circuiti. In questo modo il dato in ingresso ad un circuito della cascata è elaborato in un tempo breve e passato al circuito seguente, mentre il primo circuito può iniziare l’elaborazione del dato successivo. Procedendo in questa maniera, a parte un ritardo iniziale, si ottiene una velocità di processamento elevata, che è limitata solo dall’operazione più complessa della sequenza di operazioni invece che dall’intera elaborazione.

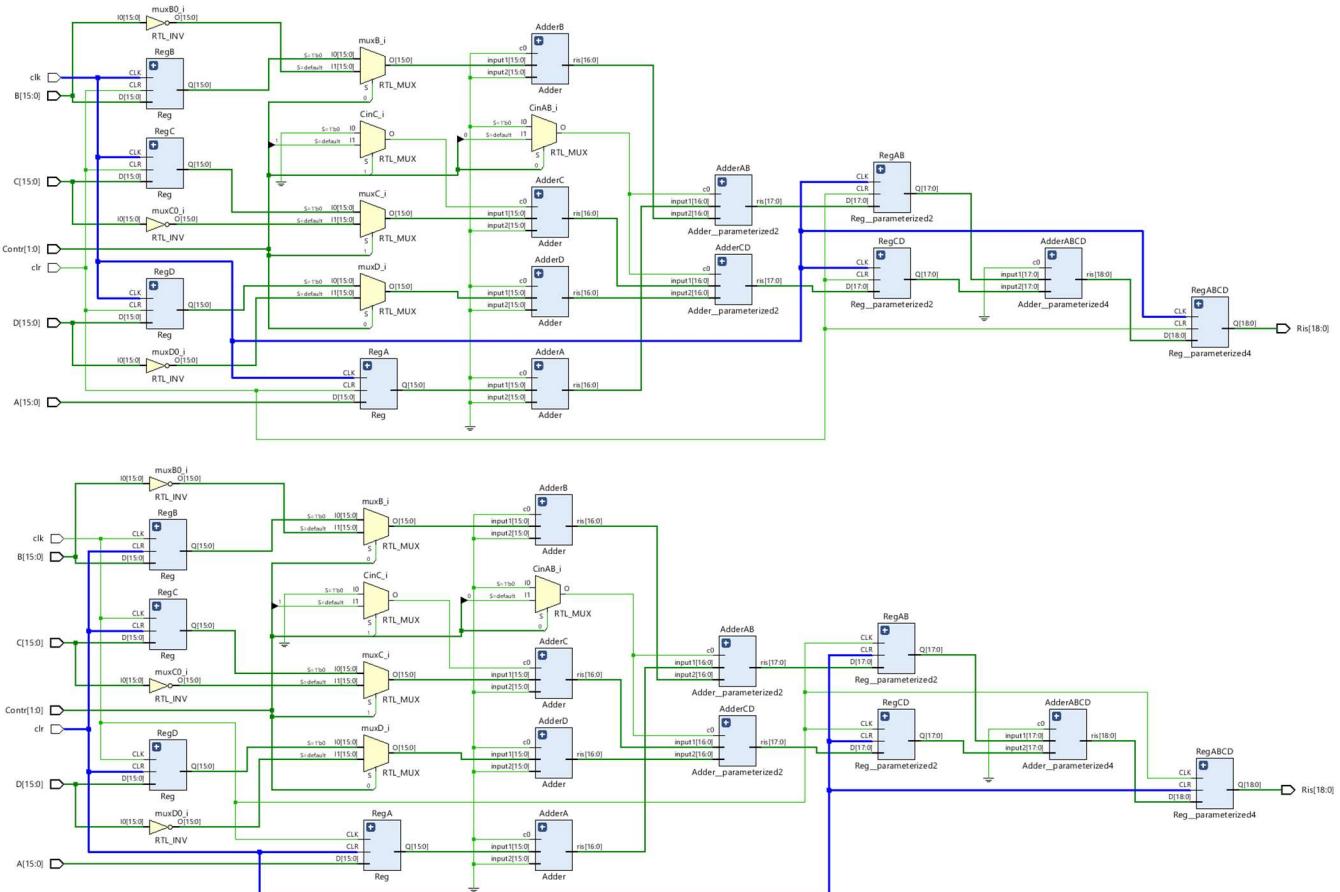
RTL Analysis – Schematic

Effettuando la *RTL-Analysis* otteniamo il seguente *Schematic* per il circuito in questione:



Posso notare come lo *Schematic* ottenuto rispecchia perfettamente lo schema già descritto precedentemente (ovviamente lo *Schematic* ottenuto tramite *RTL Analysis* include i segnali di *clock*, *clear* e di controllo *Contr* che non avevo rappresentato precedentemente per non appesantire lo schema).

Evidenzio, inoltre, rispettivamente i segnali di *clock* e di *clear* comuni a tutti i registri del circuito:



PipelineCircuit4Operands Simulation

Il seguente codice vhdl rappresenta il file di testbench utilizzato per effettuare le varie simulazioni del circuito. Ho deciso di fare una simulazione esaustiva, cioè una simulazione che ricopre tutti i possibili numeri rappresentabili con i bit corrispondenti. Ho considerato un segnale di controllo pari a “11”, cioè tale che l’operazione svolta dal circuito sarà $A-B-C-D$. Nella *begin* dell’*Architecture* corrispondente ho inizializzato due process, il primo per impostare la ciclicità del clock con un periodo di $12ns$ tale che ogni transizione da ‘0’ a ‘1’ viene effettuata ogni $6ns$, mentre nel secondo process viene effettuata la vera e propria simulazione dove inizialmente applico un’attesa al circuito pari alla somma tra il periodo del clock e $100ns$, poichè altrimenti rischieremmo che l’output del circuito non segua le variazioni degli input dello stesso. Infatti, se non mettessimo questa attesa, potremmo vedere dalla simulazione che tale fenomeno termina intorno ai $100ns$ proprio perché il dispositivo automaticamente attiva un segnale di *global reset* che tiene *bloccati* i registri proprio per $100ns$. Pertanto, per tenere conto di questa *pausa*, introduco un’attesa pari a $Tclk+100ns$ tale per cui le operazioni verranno ritardate. Inoltre, considero anche un clear alto e poi di nuovo basso. Successivamente per effettuare la simulazione esaustiva instanzio 4 cicli for, ognuno per ogni input del circuito. Ovviamente nell’ultimo ciclo for, cioè quello in riferimento all’input D, instanzio 4 *statement if* proprio perché per ogni risultato ottenuto ne calcolo il *valore vero* e, inoltre, ne calcolo lo scarto di errore tra il valore ottenuto dal circuito ed il valore vero. Tali *statement* servono a gestire le 4 tipologie di operazioni possibili con il segnale di controllo Contr. Ovviamente, per ogni risultato ottenuto ad ogni iterazione di ogni ciclo for introduco un’attesa pari ad un periodo di clock tale da uniformare i risultati ottenuti per ogni ciclo di clock.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_arith.ALL;
4 library work;
5 use work.MyDefinitions.all;
6
7 entity SimPC4O is
8     generic(n:integer:=n_bit);
9 end SimPC4O;
10
11 architecture Behavioral of SimPC4O is
12     component PC4O is
13         port(
14             A,B,C,D: in std_logic_vector(n-1 downto 0);
15             Contr: in std_logic_vector(1 downto 0);
16             clk,clr: in std_logic;
17             Ris: out std_logic_vector(n+2 downto 0)
18         );
19     end component;
20     signal IA,IB,IC,ID: std_logic_vector(n-1 downto 0);
21     signal Icontr: std_logic_vector(1 downto 0);
22     signal Iclk,Iclr: std_logic:='0';
23     signal ORis: std_logic_vector(n+2 downto 0);
24     constant Tclk:Time:=12ns;
25
26     signal TrueRis,Error:integer;
27
28 begin
29
30     Icontr(1)<='1';
31     IContr(0)<='1';
32
33     PC4O_forSimulation: PC4O port map(IA,IB,IC,ID,Icontr,Iclk,Iclr,ORis);
34
35
36     process
37         begin
38             wait for Tclk/2;
39             Iclk<=not Iclk;
40     end process;

```

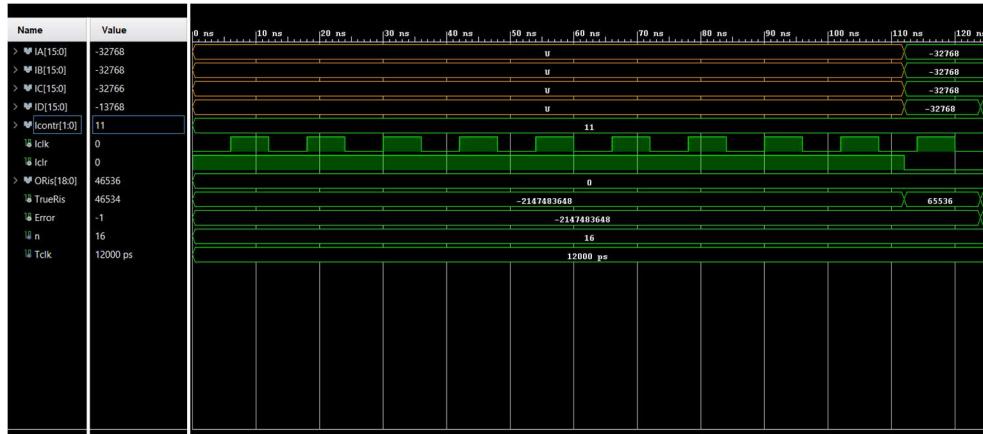
```

42 ⊕      process
43       begin
44         Iclr <='1';
45         wait for Tclk+100ns;
46         Iclr<='0';
47
48 ⊕         for va in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
49             IA <= conv_std_logic_vector( va, n );
50             for vb in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
51                 IB <= conv_std_logic_vector( vb, n );
52                 for vc in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
53                     IC <= conv_std_logic_vector( vc, n );
54                     for vd in -( 2** (n-1) ) to ( 2** (n-1)-1 ) loop
55                         ID <= conv_std_logic_vector( vd, n );
56                         if(Icontr="00")then
57                             TrueRis <= va+vb+vc+vd;
58                             Error <= TrueRis - conv_integer( signed(ORis) );
59                         elsif(Icontr="01" )then
60                             TrueRis <= va-vb+vc-vd;
61                             Error <= TrueRis - conv_integer( signed(ORis) );
62                         elsif(Icontr="10" )then
63                             TrueRis <= va+vb-vc+vd;
64                             Error <= TrueRis - conv_integer( signed(ORis) );
65                         elsif(Icontr="11" )then
66                             TrueRis <= va-vb-vc-vd;
67                             Error <= TrueRis - conv_integer( signed(ORis) );
68                         end if;
69                         wait for Tclk;
70                         end loop;
71                         wait for Tclk;
72                         end loop;
73                         wait for Tclk;
74                         end loop;
75                         wait for Tclk;
76                         end loop;
77         end process;
78
79 ⊕ end Behavioral;

```

Behavioral Simulation

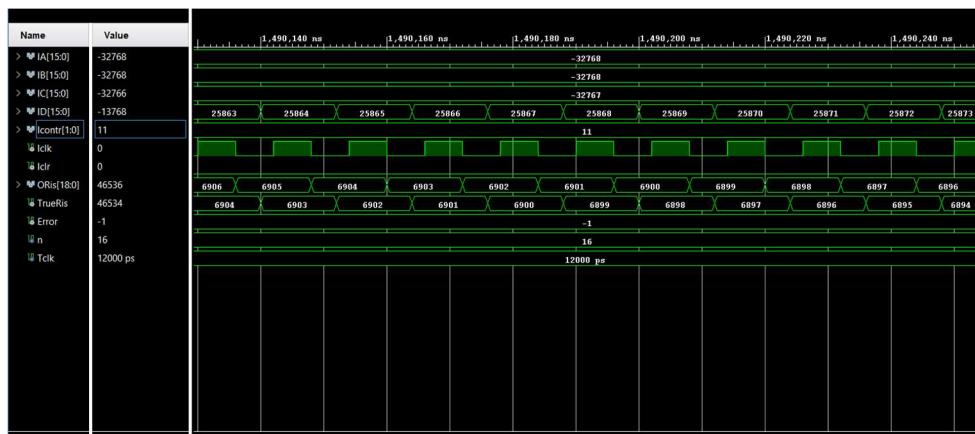
Allego gli screenshot relativi alla Behavioral Simulation del circuito in questione.



Possiamo notare gli effetti del clear alto inizialmente e poi successivamente di nuovo basso ed un ritardo iniziale pari a $T_{clk}+100ns$ tale per cui tutti gli ingressi del circuito rimangono *Uninitialized* ('U').



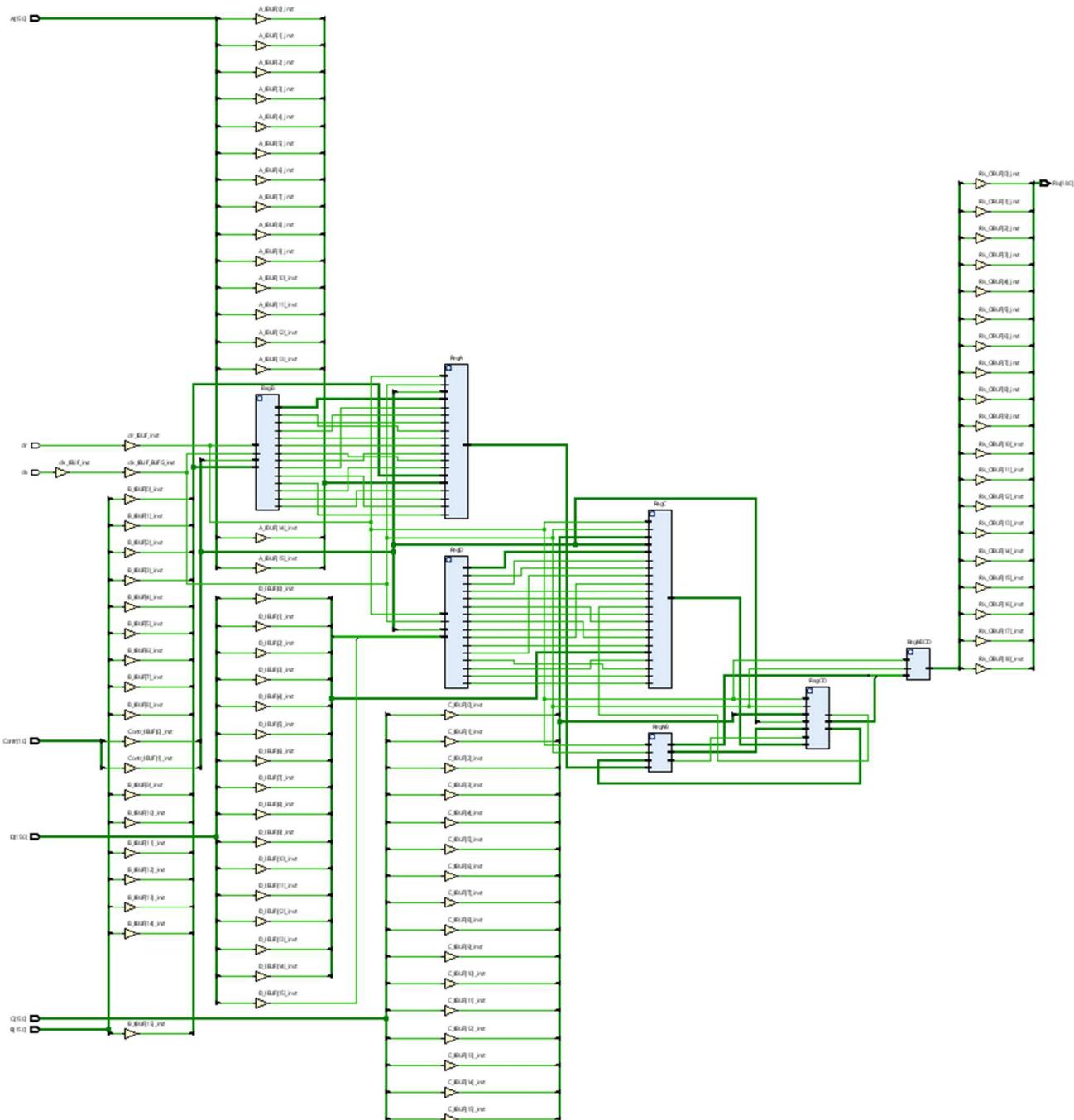
Possiamo notare come ad ogni fronte di discesa del clock otteniamo dal circuito un risultato tale per cui lo scarto di errore tra il valore vero e il valore ottenuto dal circuito stesso è pari a 1 (dovuto al fatto che il valore vero risulta non avere alcun ritardo).



Synthesis

Synthesis – Schematic

Allego lo *Schematic* relativo alla sintesi del circuito in questione.

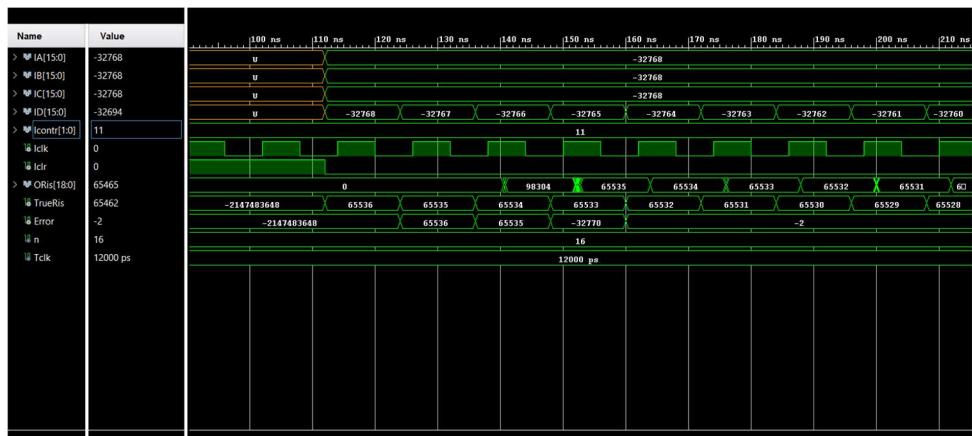


Post-Synthesis Timing Simulation

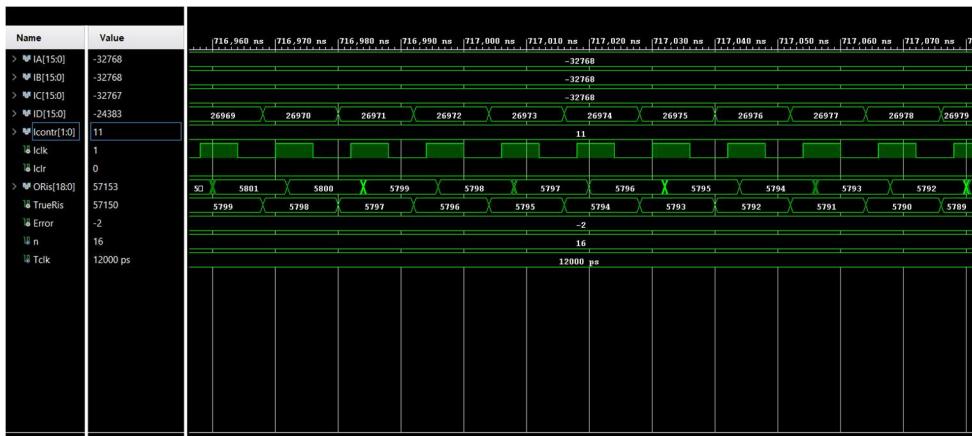
Allego gli screenshot relativi alla Post-Synthesis Timing Simulation del circuito in questione.



Possiamo notare inizialmente una non specificazione 'X' dovuto al fatto che il software di simulazione sta ancora elaborando gli ingressi. Tutto ciò è dovuto al ritardo dei componenti.



Possiamo notare un opportuno ritardo dovuto alla naturale evoluzione del clock tale per cui successivamente avremo gli effettivi risultati del circuito. Ovviamente, ogni risultato ottenuto avrà un certo ritardo dovuto proprio ai ritardi introdotti dalla logica e dalle interconnessioni del circuito in questione.



Post-Synthesis Report

Allego il report relativo al processo di sintesi applicato al circuito in questione caratterizzato a 16-bit.

Utilization		Post-Synthesis Post-Implementation	
		Graph Table	
Resource	Estimation	Available	Utilization %
LUT	129	53200	0.24
FF	118	106400	0.11
IO	87	200	43.50
BUFG	1	32	3.13

Possiamo notare come le LUT relative al circuito in questione hanno un fattore di utilizzazione pari al 0.24% rispetto alle LUT complessive che sono pari a 53200. Inoltre, possiamo notare che i Flip Flop utilizzati sono 118 rispetto ai 106400 disponibili. Oltre tutto, sono state utilizzate il 43.50% delle risorse Input/Output disponibili e il 3.13% dei Global Buffer² del device.

Per un circuito ad 8-bit, invece, avremo il seguente report:

Utilization		Post-Synthesis Post-Implementation	
		Graph Table	
Resource	Estimation	Available	Utilization %
LUT	57	53200	0.11
FF	62	106400	0.06
IO	47	200	23.50
BUFG	1	32	3.13

Possiamo notare come le risorse utilizzate si sono più che dimezzate. Infatti, vengono utilizzate solo lo 0.11% delle LUT e lo 0.06% dei Flip Flop. Inoltre, vengono utilizzate soltanto il 23.50% delle risorse Input/Output.

² I buffer globali sono comunemente usati per le reti di clock tale da fornire la minor quantità di disallineamento possibile tra i registri che si trovano fisicamente a grandi distanze l'uno dall'altro. È inoltre possibile utilizzarli per fornire un accesso rapido ai segnali di controllo nelle applicazioni ad alta velocità.

Constraint

Introduco un constraint temporale cliccando sulla voce *Edit Timing Constraints* e poi successivamente sulla voce *Create Clock*. Nella finestra aperta bisognerà specificare un *Clock name*, cioè un nome astratto per il segnale in questione, e *Source Objects*, cioè la sorgente da cui prendere il segnale in questione. Aprendo l'ulteriore finestra, potrò, tramite la sezione della voce *Nets*, cercare tra i vari collegamenti il segnale in questione, cioè *clk* (clock). Lo seleziono e lo sposto nella finestra dei termini selezionati per indicare che questo è il segnale su cui voglio applicare il constraint. Si può notare che di default la *Waveform* ha periodo pari a *10ns* tale che il segnale del clock avrà un *duty cycle del 50%* a prescindere dei valori che caratterizzano la *Waveform*. Modifico il periodo a *12ns* e, infine, salvo il file di constraint che sarà aggiunto al progetto in questione.

```
1 | create_clock -period 12.000 -name myclock -waveform {0.000 6.000} [get_nets clk]
```

```
|-----|  
| Clock Summary  
| -----|  
|-----|
```

Clock	Waveform(ns)	Period(ns)	Frequency (MHz)
myclock	{0.000 6.000}	12.000	83.333

3

Posso notare il comando *create_clock* che verrà interpretato dal sintetizzatore e successivamente in fase di implementazione come vincolo sul periodo di clock. Pertanto, ripetiamo la sintesi e l'implementazione dato che ora è presente questo nuovo file di constraint.

³ Il constraint a *12ns* corrisponde ad una frequenza di *83.333MHz*.

Per verificare che il constraint è stato applicato correttamente analizzo il report presente nei report di *Route Design*, cioè *impl1_route_report_timing_summary_0*. In tale file trovo descritte tutte le caratteristiche temporali del mio circuito. Infatti, posso notare la voce *Design Timing Summary* dove si descrive il tempo di calcolo del circuito in questione.

```
| Design Timing Summary
| -----
```

WNS (ns)	TNS (ns)	TNS Failing Endpoints	TNS Total Endpoints
3.938	0.000	0	54

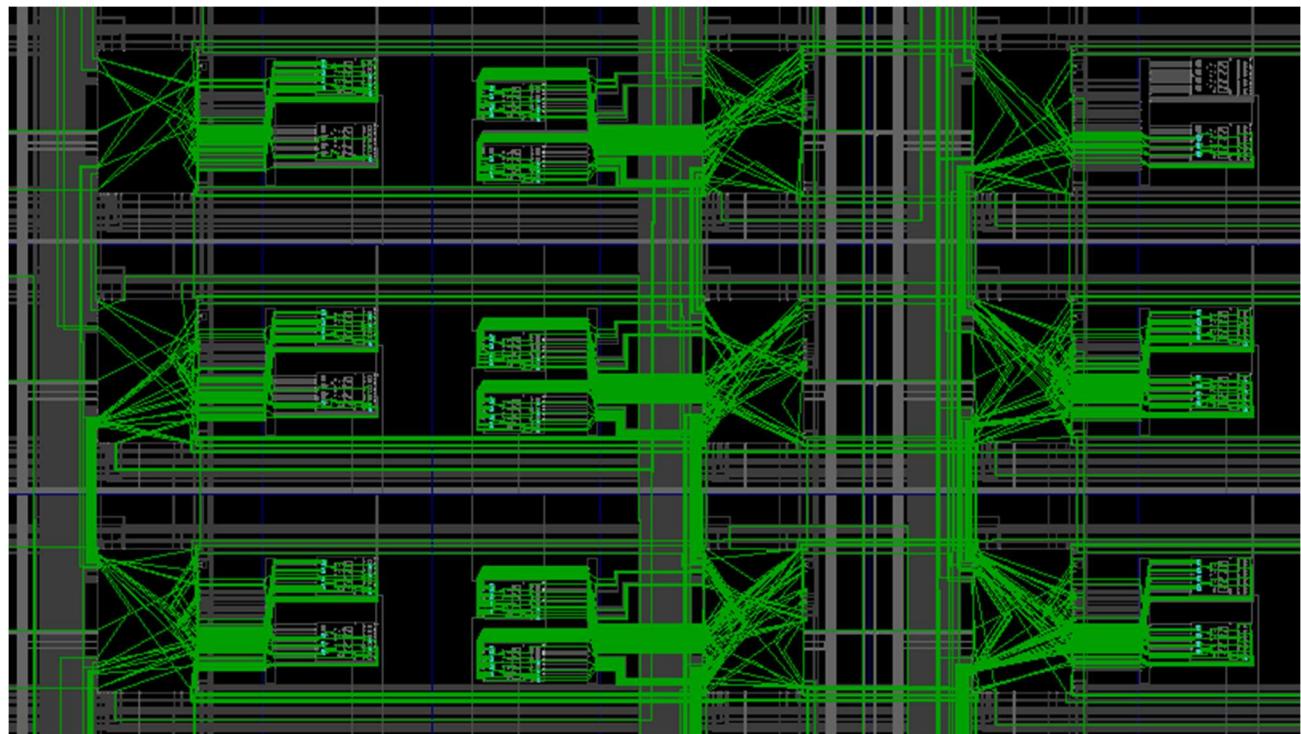
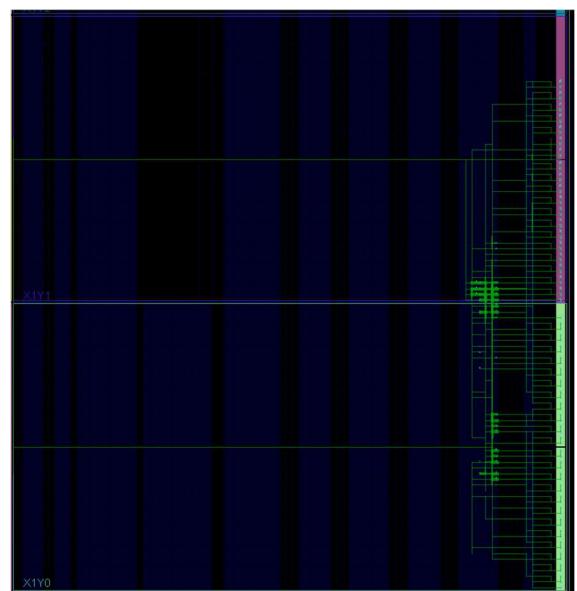
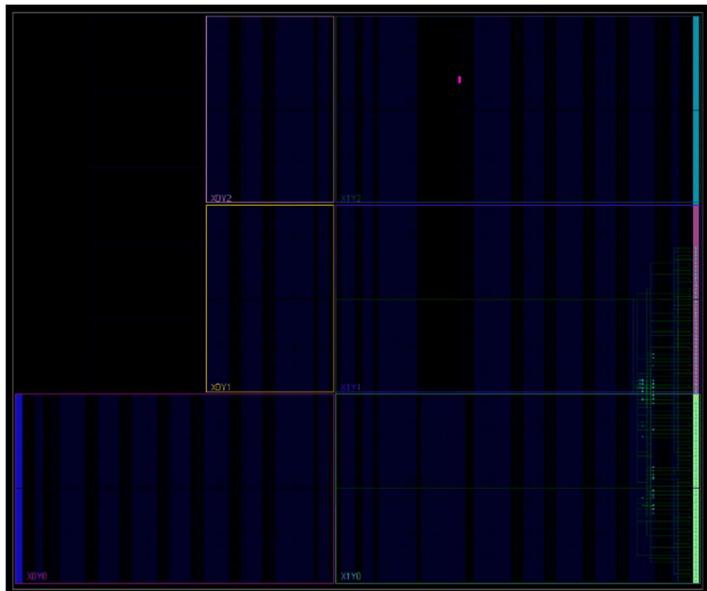
WHS (ns)	THS (ns)	THS Failing Endpoints	THS Total Endpoints
0.118	0.000	0	54

Il *WNS(ns)* identifica il *Worst Negative Slack*, cioè un valore che se positivo rappresenta il fatto che il circuito può lavorare con una frequenza superiore rispetto a quella fissata. Nel mio caso ho scelto 12ns ma il periodo di clock per questo circuito può essere anche $12\text{ns}-3.938\text{ns}=8.062\text{ns}$ tale da far lavorare il circuito più rapidamente. Possiamo notare, inoltre, che il constraint è stato applicato correttamente: *All user specified timing constraints are met*. Nel caso di periodi di clock troppo piccoli specificati nel constraint, avremmo una segnalazione nella quale viene specificata che ci sono percorsi di calcolo che non consentono il rispetto del vincolo specificato nel constraint stesso.

Implementation

Implementation – Device

Una volta sintetizzato il circuito posso procedere con l'implementazione del circuito su Device tramite *Run Implementation*. Nelle successive immagini si può notare come il circuito viene implementato su Device e, soprattutto, come tra le migliaia di LUT disponibili, ne vengono utilizzate un numero piccolissimo.

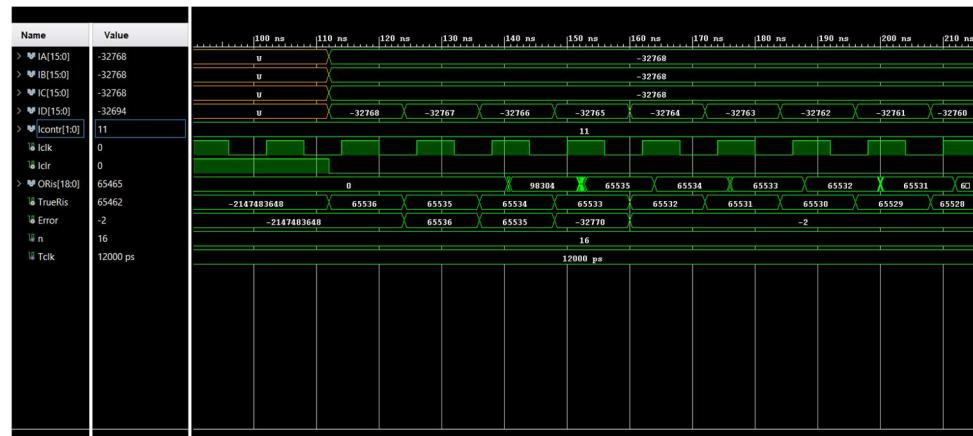


Post-Implementation Timing Simulation

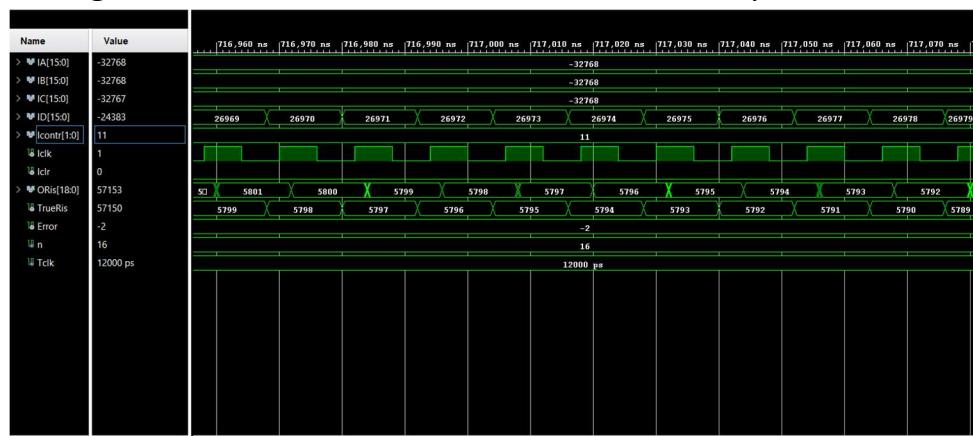
Allego gli screenshot relativi alla Post-Implementation Timing Simulation del circuito in questione.



Possiamo notare inizialmente una non specificazione 'X' dovuto al fatto che il software di simulazione sta ancora elaborando gli ingressi.



Possiamo notare un opportuno ritardo dovuto alla naturale evoluzione del clock. Ovviamente, ogni risultato ottenuto avrà un certo ritardo dovuto proprio ai ritardi introdotti dalla logica e dalle interconnessioni del circuito in questione.



Post-Implementation Report

Allego il report relativo al processo di implementazione applicato al circuito in questione caratterizzato a 16-bit.

Utilization		Post-Synthesis Post-Implementation	
		Graph Table	
Resource	Utilization	Available	Utilization %
LUT	129	53200	0.24
FF	118	106400	0.11
IO	87	200	43.50
BUFG	1	32	3.13

Possiamo notare come le LUT utilizzate per implementare il circuito su device sono pari al 0.24% delle totali e che i Flip Flop utilizzati sono pari a 118 rispetto ai 106400 disponibili. Oltre tutto, possiamo notare che le risorse Input/Output utilizzate sono 87 e, inoltre, che è stato utilizzato un solo Global Buffer.

Per 8-bit, invece, avremo il seguente report:

Utilization		Post-Synthesis Post-Implementation	
		Graph Table	
Resource	Utilization	Available	Utilization %
LUT	57	53200	0.11
FF	62	106400	0.06
IO	47	200	23.50
BUFG	1	32	3.13

Possiamo notare come le risorse utilizzate sono più che dimezzate rispetto a quelle utilizzate nel circuito caratterizzato a 16-bit. Infatti, le LUT utilizzate sono lo 0.11% le totali del device e i Flip Flop utilizzati sono lo 0.06% i totali. Inoltre, le risorse Input/Output utilizzate sono il 23.50% delle 200 disponibili nel device.

Latenza

La latenza è pari al numero di cicli di clock che ci si deve aspettare per avere il primo risultato valido del circuito. Quindi, il tempo di latenza sarà uguale al tempo necessario affinché il primo campione sia passato attraverso tutti gli stadi, quindi sarà uguale al tempo di clock moltiplicato il numero di stadi. Nel nostro caso, la latenza può essere calcolata come numero di registri che l'input generico attraversa meno uno. Quindi:

$$^4\text{latenza} = k - 1$$

Possiamo affermare che la latenza aumenta man mano che *frzioniamo* il circuito. Nel nostro caso, pertanto, la latenza sarà pari a 2 poiché il numero di registri che l'input generico dovrà attraversare affinché venga restituito l'output corrispondente sarà 3.

Throughput

Il throughput è il ritmo di risultati per ciclo di clock dopo aver aspettato un tempo di latenza pari a $k-1$. Nel nostro caso il tempo di latenza è unitario poiché otteniamo per ogni ciclo di clock precisamente un risultato, precisamente per ogni fronte di discesa del segnale.

Massima frequenza di funzionamento

La massima frequenza di funzionamento è la massima frequenza a cui il circuito può funzionare. Essa può essere identificata come il *critical path* del circuito in questione. Affinchè il circuito funzioni correttamente, bisogna garantire che i vincoli temporali siano rispettati. Per la precisione due relazioni devono essere garantite:

$$\begin{aligned} t_{\text{setup}} &\gg t_{\text{hold}} \\ T &= t_{\max} + t_{\text{setup}} \end{aligned}$$

⁴ k è il numero di registri che l'input generico deve attraversare affinché venga restituito l'output corrispondente a tale input.

Nel caso di un circuito a 16-bit, abbiamo un t_{setup} e un t_{hold} rispettivamente pari a:

Timing	Setup Hold Pulse Width
Worst Negative Slack (WNS):	3.938 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	54

5

Timing	Setup Hold Pulse Width
Worst Hold Slack (WHS):	0.118 ns
Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	54

6

Possiamo notare che la disegualanza tra t_{setup} e un t_{hold} è garantita. Per quanto riguarda l'uguaglianza, possiamo ricavarne il periodo massimo di funzionamento:

$$t_{max} = T - t_{setup}$$

Esso sarà pari a $t_{max} = 12\text{ns} - 3.938\text{ns} = 8.062\text{ns}$. Pertanto, la massima frequenza di funzionamento del circuito sarà pari a:

$$f_{max} = \frac{1}{t_{max}} = \frac{1}{8.062\text{ns}} = \frac{1}{8.062 \times 10^{-9} \text{s}} \cong 124038700 \text{ Hz}$$

5 Setup Area (Max Delay Analysis)

- Il Worst Negative Slack (WNS) corrisponde al peggior slack di tutti i percorsi di temporizzazione per l'analisi del ritardo massimo. Può essere positivo o negativo.
- Il Total Negative Slack (TNS) è la somma di tutte le violazioni WNS, se si considera solo la peggiore violazione di ciascun endpoint del percorso di temporizzazione. Il suo valore può essere:
 - 0 ns quando tutti i vincoli di temporizzazione sono soddisfatti per l'analisi del ritardo massimo
 - negativo quando ci sono delle violazioni
- Il Number of Failing Endpoints è il numero totale di endpoint con una violazione ($WNS < 0 \text{ ns}$)
- Il Total Number of Endpoints è il numero totale di endpoint analizzati.

6 Hold Area (Min Delay Analysis)

- Il Worst Hold Slack (WHS) corrisponde al peggior slack di tutti i percorsi di temporizzazione per l'analisi del ritardo minimo. Può essere positivo o negativo.
- Il Total Hold Slack (THS) è la somma di tutte le violazioni WHS, considerando solo la peggiore violazione di ciascun endpoint del percorso temporale. Il suo valore può essere:
 - 0 ns quando tutti i vincoli temporali sono soddisfatti per l'analisi del ritardo minimo
 - negativo quando ci sono delle violazioni
- Il Number of Failing Endpoints è il numero totale di endpoint con una violazione ($WHS < 0 \text{ ns}$)
- Il Total Number of Endpoints è il numero totale di endpoint analizzati.

Quindi, la massima frequenza di funzionamento del circuito a 16-bit sarà pari a

$$f_{maxMHz} = \frac{f_{maxHz}}{1 \times 10^6} = \frac{124038700}{1000000} \cong 124.0387 \text{ MHz}$$

che ovviamente risulta essere maggiore della frequenza di funzionamento garantita dal vincolo di constraint pari a **83.333 MHz**.

Nel nostro caso dato che *WNS* e il *WHS* sono entrambi positivi e, pertanto, il numero di endpoint con una violazione è pari a zero su un'analisi di 54 endpoint. Inoltre, il *TNS* e il *THS* sono entrambi pari a *0 ns* proprio perché i vincoli di temporizzazione sono soddisfatti rispettivamente per l'analisi del ritardo massimo e per l'analisi del ritardo minimo.

Se consideriamo, invece, il circuito ad 8-bit allora avremo i seguenti parametri:

Timing	Setup		Hold		Pulse Width
Worst Negative Slack (WNS):	7.353 ns				
Total Negative Slack (TNS):	0 ns				
Number of Failing Endpoints:	0				
Total Number of Endpoints:	30				

Timing	Setup		Hold		Pulse Width
Worst Hold Slack (WHS):	0.129 ns				
Total Hold Slack (THS):	0 ns				
Number of Failing Endpoints:	0				
Total Number of Endpoints:	30				

Possiamo notare che la disegualanza tra t_{setup} e un t_{hold} è garantita. Per quanto riguarda l'uguaglianza, possiamo ricavarne il periodo massimo di funzionamento:

$$t_{max} = T - t_{setup}$$

Esso sarà pari a $t_{max} = 12ns - 7.353ns = 4.647ns$. Pertanto, la massima frequenza di funzionamento del circuito sarà pari a:

$$f_{max} = \frac{1}{t_{max}} = \frac{1}{4.647ns} = \frac{1}{4.647 \times 10^{-9}s} \cong 215192597.4 \text{ Hz}$$

Quindi, la massima frequenza di funzionamento del circuito a 16-bit sarà pari a

$$f_{maxMHz} = \frac{f_{maxHz}}{1 \times 10^6} = \frac{215192597.4}{1000000} \cong 215.1925 \text{ MHz}$$

che ovviamente risulta essere maggiore della frequenza di funzionamento garantita dal vincolo di constraint pari a **83.333 MHz**.

Power On-Chip

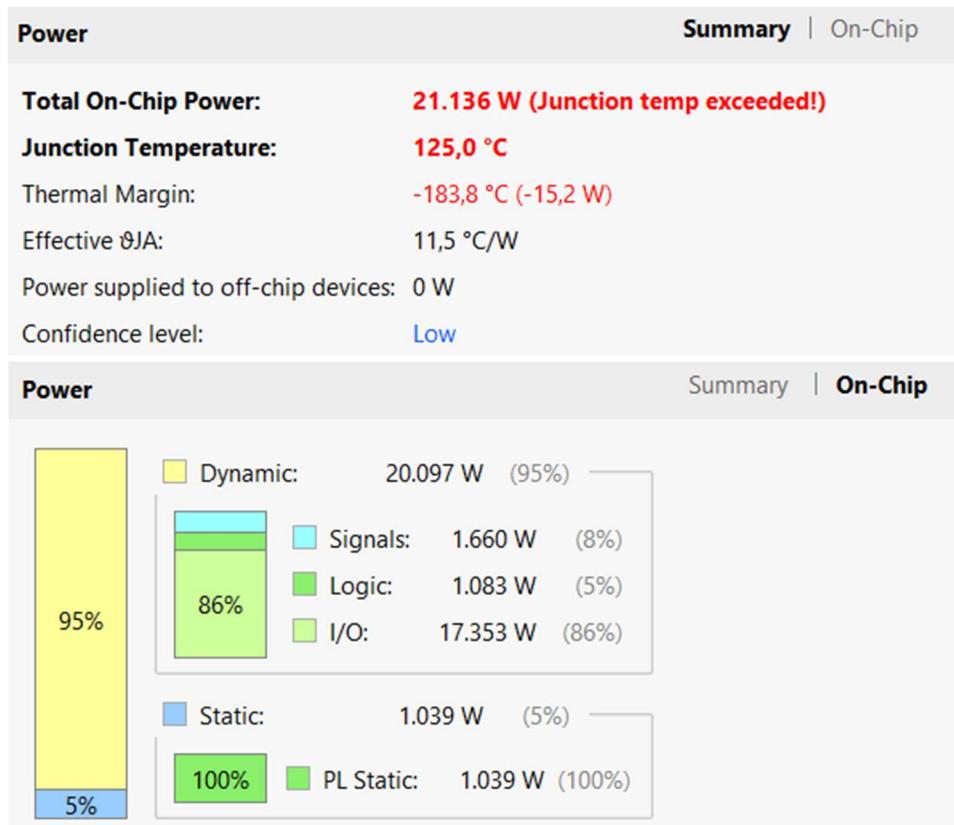
La potenza statica di un *device* è la potenza della dispersione del transistor su tutte le linee di tensione collegate e sui circuiti necessari per il normale funzionamento dell'*FPGA*, dopo la configurazione. Questa rappresenta lo stato stazionario, cioè la perdita intrinseca del *device*. La potenza dinamica di un *device*, invece, è dovuta al modello di dati di input e all'attività interna del design. Questa potenza è instantanea e varia ad ogni ciclo di clock. Dipende dai livelli di tensione e dalle risorse logiche e di routing utilizzate. Ciò include anche la corrente statica delle terminazioni I/O, dai gestori di clock e da altri circuiti che necessitano di alimentazione quando utilizzati. Essa non include l'alimentazione fornita ai dispositivi off-chip. Inoltre, la potenza totale su chip (*Total On-Chip Power*) è la potenza consumata internamente all'interno dell'*FPGA*, pari alla somma della *Static Power* e della *Dynamic Power* del *device*. Essa è anche conosciuta come potenza termica. L'*Ambient Temperature* è la temperatura dell'aria che circonda immediatamente il dispositivo nelle condizioni operative previste dal sistema. L'*Effective Thermal Resistance to Air (θ_{JA})* è un coefficiente che definisce come la potenza viene dissipata dal silicio del *FPGA* all'ambiente (giunzione del device all'aria ambiente). Infine, la temperatura di giunzione (*Junction Temperature*) è la temperatura del dispositivo in funzione. In genere, quando si seleziona il dispositivo, si sceglie un grado di temperatura. Questo grado definisce un intervallo di temperatura che garantisce il corretto funzionamento del dispositivo come specificato. Se le condizioni operative sono al di sopra del grado massimo, ma rimangono al di sotto della temperatura massima assoluta, il funzionamento del dispositivo non è più garantito. Il superamento delle condizioni operative del massimo assoluto può danneggiare il dispositivo. La temperatura di giunzione può essere calcolata nel seguente modo:

$$JT = AT + TOCP * ETRA(\theta_{JA})$$

dove:

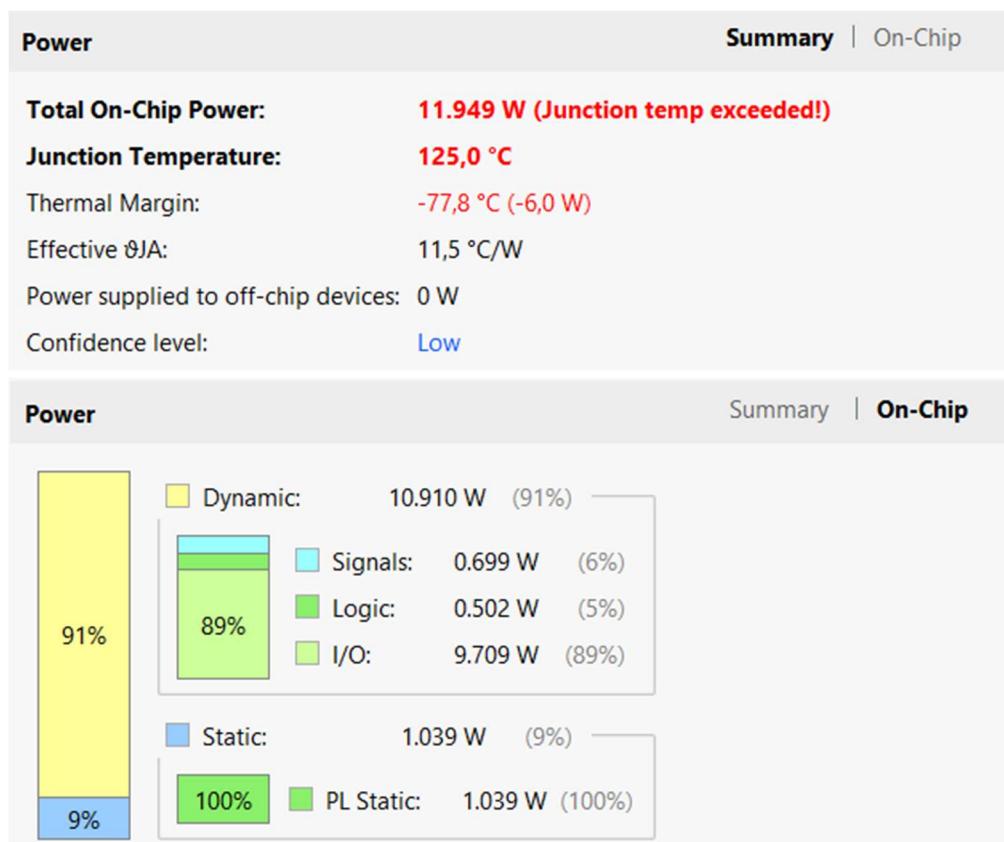
- ***JT*** è la Junction Temperature
- ***AT*** è l'Ambient Temperature
- ***TOCP*** è la Total On-Chip Power
- ***ETRA(θ_{JA})*** è LA Effective Thermal Resistance to Air

Prima di applicare il file di constraint avremo una dissipazione di potenza per un circuito a 16-bit del tipo:



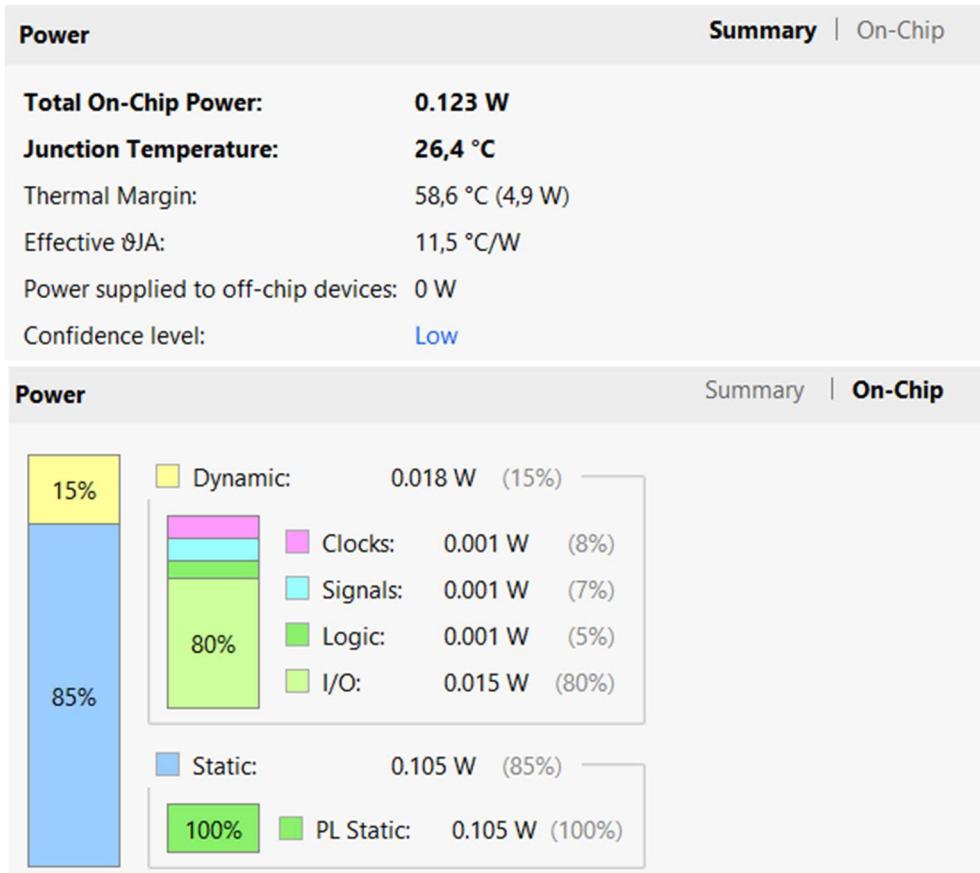
Possiamo notare una segnalazione riguardo la dissipazione di potenza e riguardo la temperatura di giunzione. Pertanto, viene applicato un constraint tale da imporre un vincolo al circuito stesso ed eliminare i problemi sopra citati.

Prima di applicare il file di constraint avremo una dissipazione di potenza per un circuito a 8-bit del tipo:



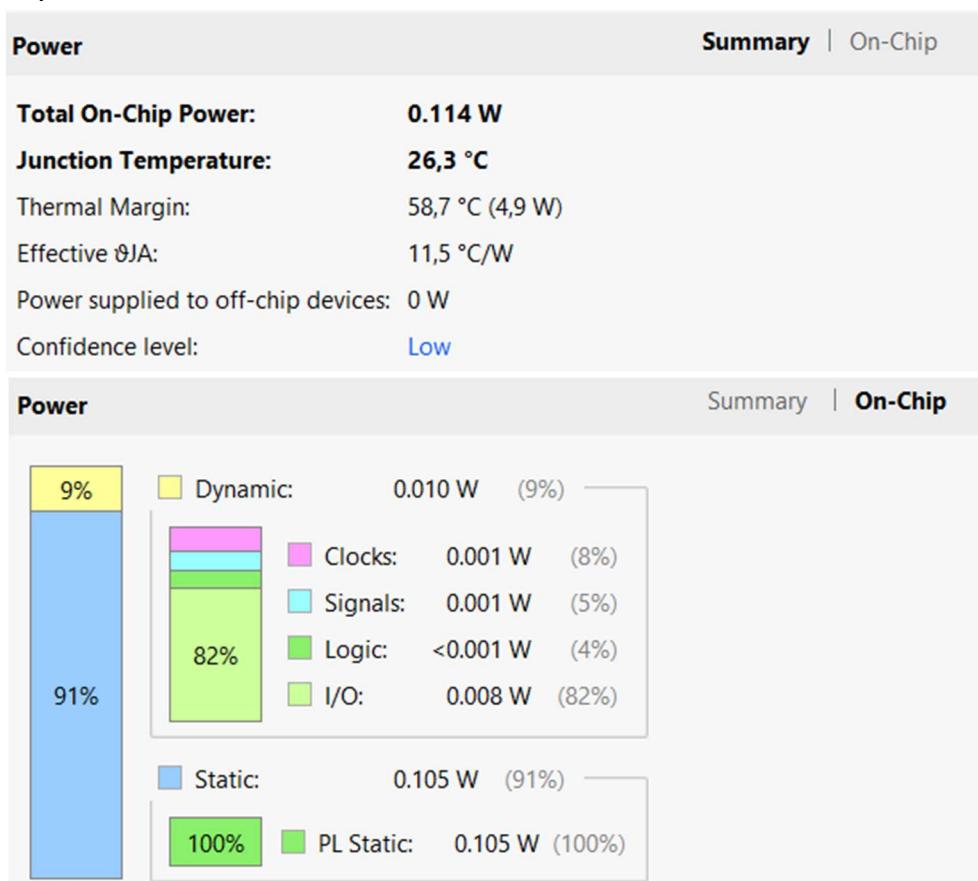
Ovviamente anche in questo avremo una segnalazione riguardo la dissipazione di potenza e riguardo la temperatura di giunzione.

Dopo aver applicato il file di constraint avremo una dissipazione di potenza per un circuito a 16-bit del tipo:



Possiamo notare che il problema per quanto riguarda la temperatura di giunzione è scomparso. Infatti, la dissipazione di potenza è stata stimata alla frequenza di funzionamento che corrisponde ai 12ns definiti nel file di constraint. Possiamo notare che la dissipazione statica è di gran lunga superiore a quella dinamica. Infatti, analizzando il *device* possiamo constatare che una piccola parte di esso viene utilizzata e le sole risorse utilizzate sono responsabili della dissipazione dinamica. Tutto il resto, cioè il circuito non utilizzato, è responsabile della dissipazione statica. Quindi, ci sono molte risorse che dissipano staticamente rispetto a quelle che dissipano dinamicamente perché stanno effettivamente lavorando. Inoltre, possiamo notare che dopo aver applicato il file di constraint, è apparso il contributo legato al segnale di Clock. Tale segnale è molto particolare perché, dato che deve arrivare a tanti Flip Flop, è necessario gestire il suo *carico* in una maniera particolare. Proprio per questo motivo la distribuzione del segnale di clock è responsabile di una quota considerevole di dissipazione di potenza dinamica ($0.001 \text{ W} - 8\%$).

Invece, applicando il file di constraint ad un circuito ad 8-bit avremo una dissipazione di potenza del tipo:



Possiamo notare come la potenza dinamica del device è diminuita. Questo è dovuto al fatto che le risorse utilizzate sono quasi praticamente la metà rispetto alle risorse utilizzate nel device caratterizzato a 16-bit. Anche i segnali generici, la logica del circuito e le risorse Input/Output dissipano di meno rispetto al circuito a 16-bit. Tanto è vero che un'ulteriore conferma si può avere osservando rispettivamente i device relativi al circuito caratterizzato a 8-bit e al circuito caratterizzato a 16-bit:

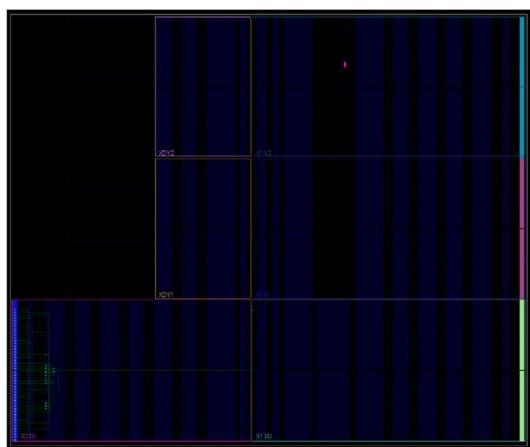


Figure 8 Device Implementation del circuito a 8-bit

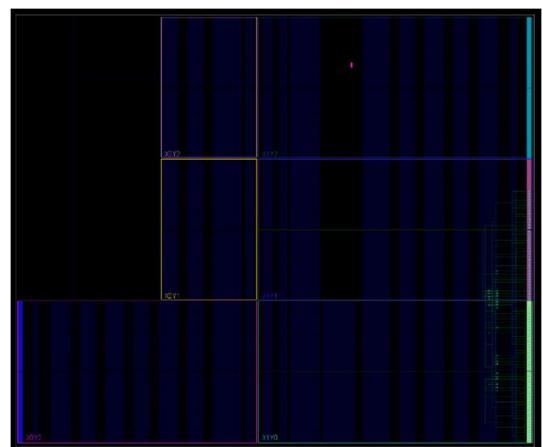


Figure 9 Device Implementation del circuito a 16-bit