

Mini ALU

Relazione di progetto

Studenti:

Frega Umberto 239527, frgmrt04a051353d@studenti.unical.it;

Napoli Leonardo 234364, np11rd02s30d086@studenti.unical.it;

Codice Sorgente

Il progetto assegnato consiste nel progettare ed implementare una mini alu, capace di fare addizioni e sottrazioni, tramite linguaggio VHDL. Per la progettazione del sistema si è deciso di utilizzare un pattern comportamentale, andando quindi a definire il comportamento del sistema in base a determinate condizioni, oltretutto si è optato per l'utilizzo del tipo *STD_LOGIC* e quindi *STD_LOGIC_VECTOR* per una maggiore flessibilità e maggiori funzionalità.

Il primo passo della progettazione è stato definire la politica tramite la quale la mini ALU potesse cambiare tra addizione e sottrazione. A questo proposito si è deciso di mantenere un singolo adder, ma cambiare il segno del secondo operando.

1 Adder

1.1 Implementazione

La componente di base del sistema è un carry look-ahead adder, che genera quindi vari segnali generate e propagate a seconda del numero di bit degli operandi. Riportiamo di seguito il codice dell'adder con caso di default con 4 bit.

1: Codice adder

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  library work;
4  use work.constants.all;
5
6  entity generic_adder is
7      generic (bit_number : INTEGER := nbit);
8      Port ( A_adder,B_adder: in STD_LOGIC_VECTOR (bit_number-1 downto 0);
9            cin : in STD_LOGIC;
10           sum : out STD_LOGIC_VECTOR (bit_number downto 0));
11 end generic_adder;
12
13 architecture Behavioral of generic_adder is
14     signal p,g : STD_LOGIC_VECTOR (bit_number downto 0);
15     signal carry : STD_LOGIC_VECTOR (bit_number+1 downto 0);
16 begin
17     carry(0) <= cin;
18     p_g: for i in 0 to bit_number generate
19         p_gMSB: if (i=bit_number) generate
20             p(i) <= A_adder(bit_number-1) xor B_adder(bit_number-1);
21             g(i) <= A_adder(bit_number-1) and B_adder(bit_number-1);
22         end generate;
23         p_gLSB: if i<bit_number generate
```

```

24         p(i) <= A_adder(i) xor B_adder(i);
25         g(i) <= A_adder(i) and B_adder(i);
26     end generate;
27     carry(i+1) <= (g(i) or (p(i) and carry(i)));
28     sum(i) <= carry(i) xor p(i);
29 end generate;
30 end Behavioral;

```

1.2 Schematica

Il codice precedente con bit number 4, 8 e 16 ha generato in vivado le schematiche riportate rispettivamente in *Figure 1*, *Figure 2*, *Figure 3*.

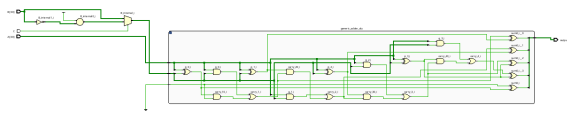


Figure 1: Adder a 4 bit

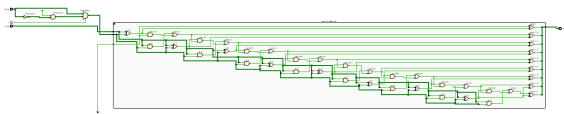


Figure 2: Adder a 8 bit

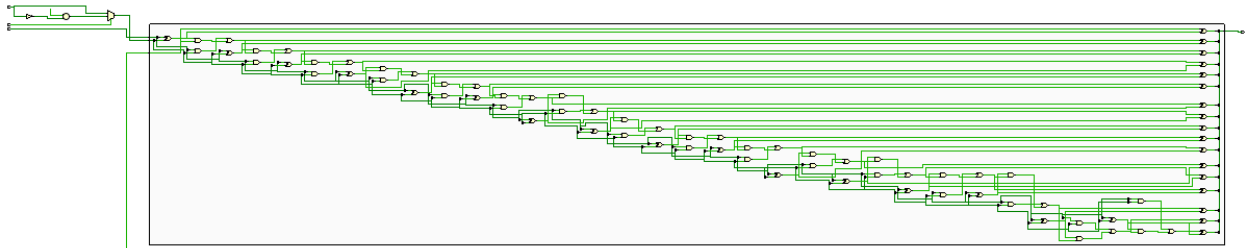


Figure 3: Adder a 16 bit

2 Mini ALU

2.1 Implementazione

La mini ALU progettata presenta al suo interno un solo adder, preceduto da un multiplexer, che in base al bit di controllo C decide se dare in output B oppure il risultato di B invertito. Nell'adder poi, oltre ad A ed al valore calcolato di B, verrà introdotto il valore di C stesso, completando il complemento a 2 in caso di necessità, non apportando cambiamenti altrimenti.

2: Codice Mini ALU

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  library work;
4  use work.constants.all;
5
6  entity mini_alu is
7      generic (bit_number : INTEGER := nbit);
8      Port ( A,B : in STD_LOGIC_VECTOR (bit_number-1 downto 0);
9            C : in STD_LOGIC;
10           output : out STD_LOGIC_VECTOR (bit_number downto 0));
11 end mini_alu;
12
13 architecture Behavioral of mini_alu is
14     component generic_adder is
15         generic (bit_number:INTEGER := nbit);
16         Port (
17             A_adder, B_adder : in STD_LOGIC_VECTOR (bit_number-1 downto 0);
18             cin : in STD_LOGIC;
19             sum : out STD_LOGIC_VECTOR (bit_number downto 0));
20     end component;
21
22     signal B_internal: STD_LOGIC_VECTOR (bit_number-1 downto 0);
23     signal carry_in: STD_LOGIC;
24
25     begin
26         process(A, B, C) begin
27             case C is
28                 when '0' =>
29                     B_internal <= B;
30
31                 when others =>
32                     B_internal <= STD_LOGIC_VECTOR(not B);
33
34             end case;
35         end process;
36
37         generic_adder_alu: generic_adder
38             GENERIC MAP (bit_number => bit_number)
39             PORT MAP (
40                 A_adder => A,
41                 B_adder => B_internal,
42                 cin => C,
43                 sum => output);
44     end Behavioral
```

Possiamo trovare la schematica risultante nella *Figure 4*

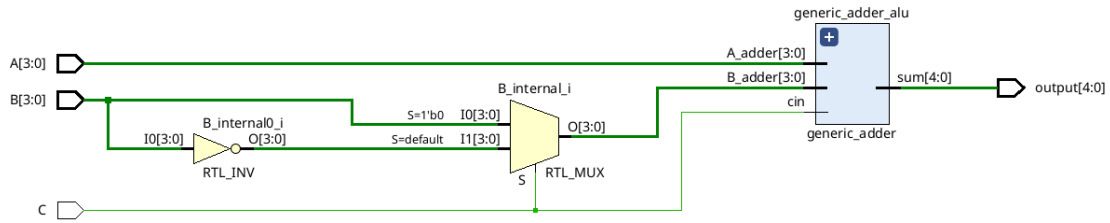


Figure 4: Circuito Logico del mini ALU

2.2 TestBench

I test sono stati svolti in tutti i casi possibili, dando un tempo di 10 ns per ogni caso, con un tempo totale in nanosecondi:

$$2^{2n+1} \times 10 \quad (1)$$

3: Codice test

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.all;
4  library work;
5  use work.constants.all;
6
7  entity mini_alu_testbench is
8      generic (bit_number: integer := nbit);
9  end mini_alu_testbench;
10
11 architecture Behavioral of mini_alu_testbench is
12     component mini_alu is
13         generic (bit_number : INTEGER := nbit);
14         Port ( A,B : in STD_LOGIC_VECTOR (bit_number-1 downto 0);
15               C : in STD_LOGIC;
16               output : out STD_LOGIC_VECTOR (bit_number downto 0));
17     end component;
18
19     constant min_value : integer := -(2**(bit_number-1));
20     constant max_value : integer := (2**(bit_number-1))-1;
21
22     signal Ia,Ib: STD_LOGIC_VECTOR (bit_number-1 downto 0);
23     signal Ic: STD_LOGIC;
24     signal Ooutput: STD_LOGIC_VECTOR(bit_number downto 0);
25 begin
26     CUT: mini_alu port map(Ia,Ib,Ic, Ooutput);
27     process
28     begin
29         external: for i in min_value to max_value loop
30             Ia <= (STD_LOGIC_VECTOR((TO_SIGNED(i,bit_number))));
31             internal: for j in min_value to max_value loop

```

```
32         Ic <= '0';
33         Ib <= (STD_LOGIC_VECTOR((TO_SIGNED(j,bit_number))));
34         wait for 10ns;
35         Ic <= '1';
36         wait for 10ns;
37     end loop internal;
38 end loop external;
39 end process;
40 end Behavioral;
```

3 Simulazione

Sono state effettuate simulazioni behavioural e post-implementation. Vediamole, evidenziandone le differenze.

3.1 4 bit

Per prime analizziamo le simulazioni nel caso del circuito a 4 bit. Per semplicità verranno riportate solo le immagini di alcuni momenti salienti della simulazione

3.1.1 Behavioural

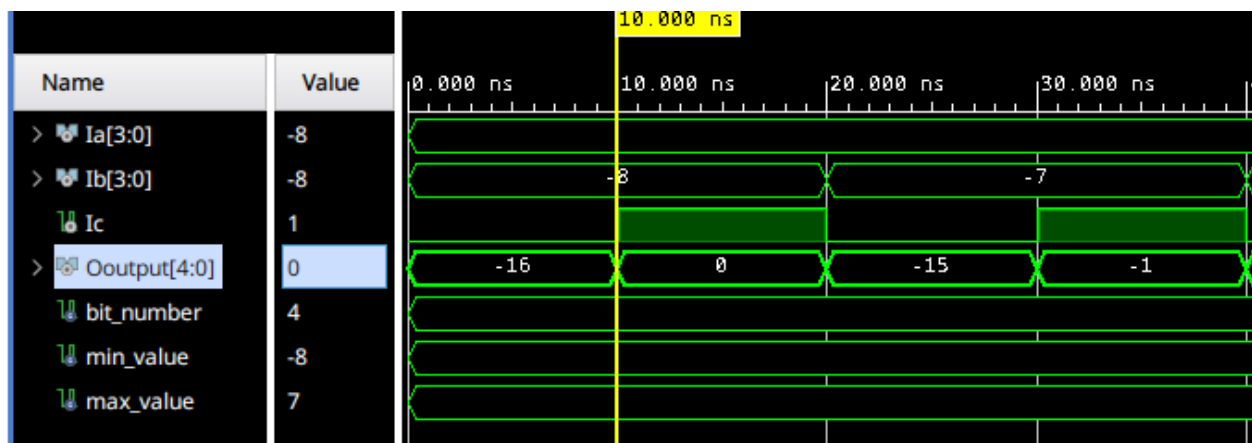


Figure 5: Cambio di Ic

Possiamo notare come la simulazione in *Figure 5* restituisca i dati corretti senza nessun delay. Vediamo ora la stessa situazione in un'implementazione reale.

3.1.2 Post-implementation

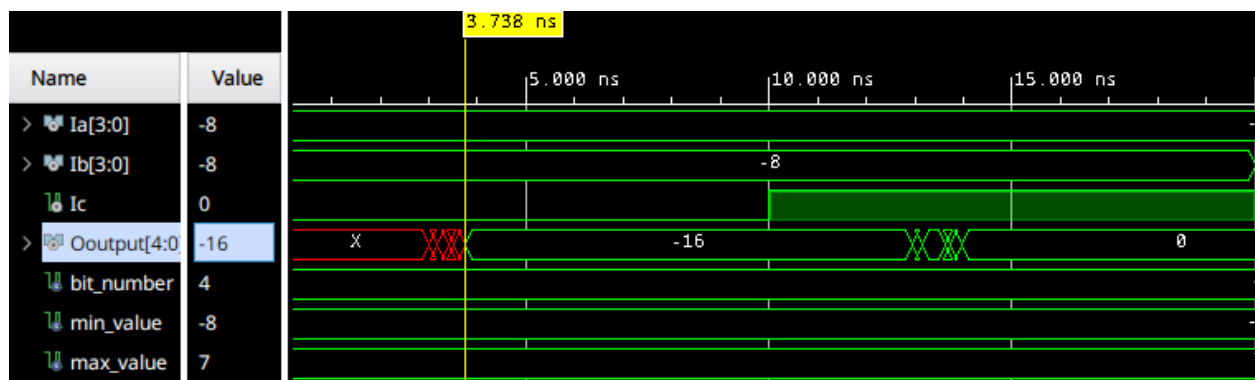


Figure 6: Fine output non definito

Degna di nota la prima differenza mostrata in *Figure 6*, nonostante gli input vengano dati al tempo iniziale 0, sono necessari 3,738 ns affinché il circuito produca il primo risultato utile.

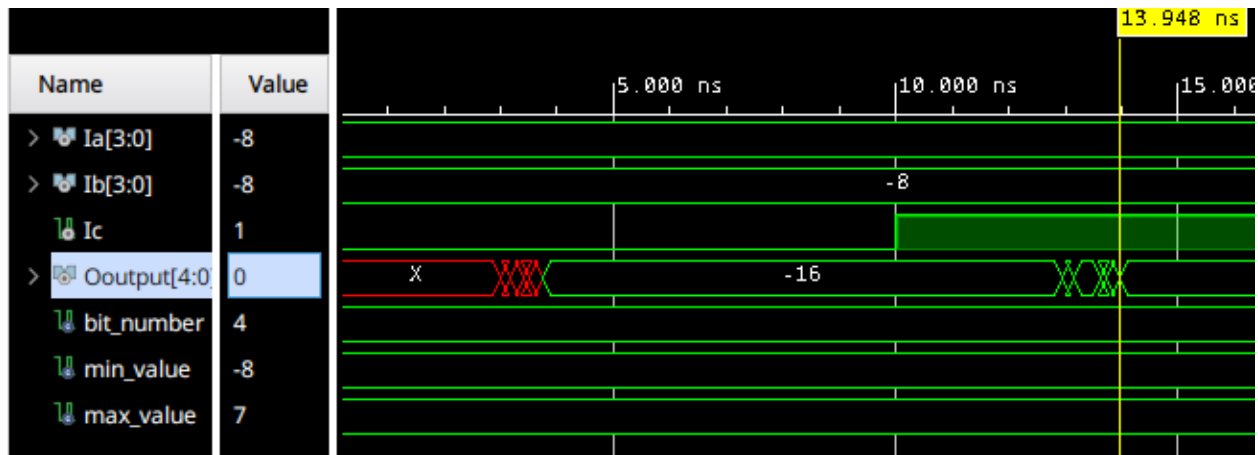


Figure 7: Cambio di C

Ancora, affinché il risultato del cambio di valore di C possa essere utilizzato sono necessari 3,948 ns, come si vede in *Figure 7*

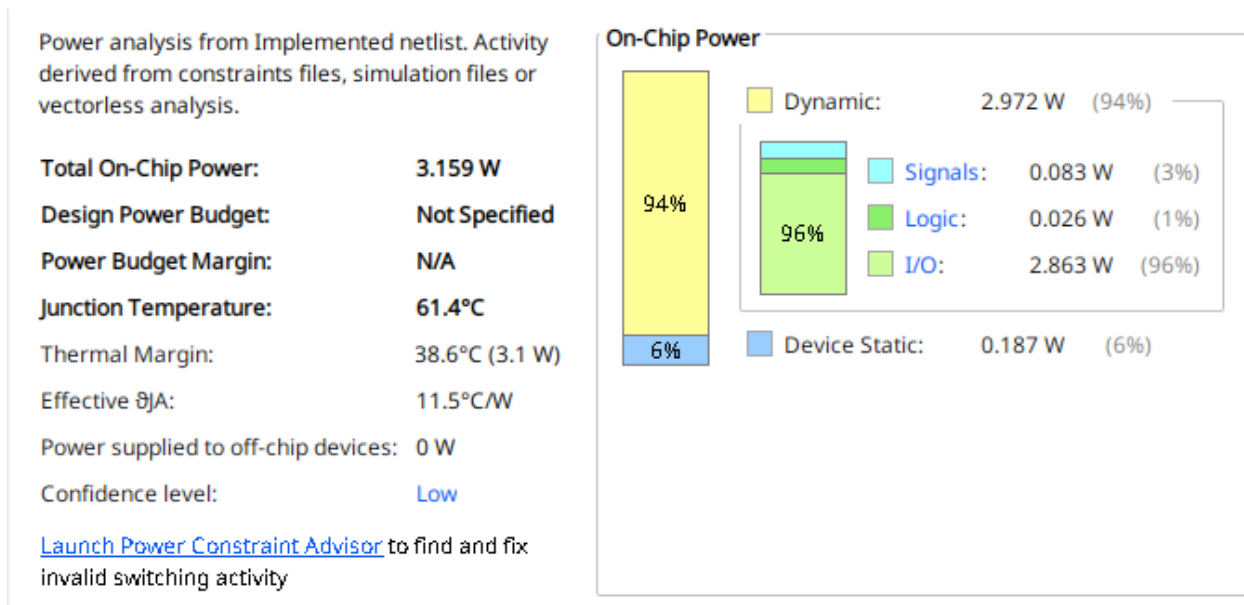


Figure 8: Junction Temperature e Total Power

Da *Figure 8* notiamo che l'implementazione del circuito, sebbene non presenti un clock, non produce problemi significativi dal punto di vista della temperatura, grazie al basso numero di bit da calcolare.

3.1.3 LUT Tables

Dalla sintesi del circuito ci vengono restituite inoltre le LUT Tables con i relativi valori di utilizzo. Il loro numero è riportato in *Table 1*, mentre in *Figure 9* troviamo la schematica del circuito reale.

Riportiamo per completezza in *Table 2* anche l'utilizzazione delle LUT, estremamente bassa a causa della semplicità del circuito

Nome	Utilizzate	Tipo di utilizzo
IBUF	9	IO
OBUF	5	IO
LUT6	2	LUT
LUT5	2	LUT
LUT4	1	LUT
LUT2	1	LUT

Table 1: Numero LUT utilizzati nel circuito a 4 bit

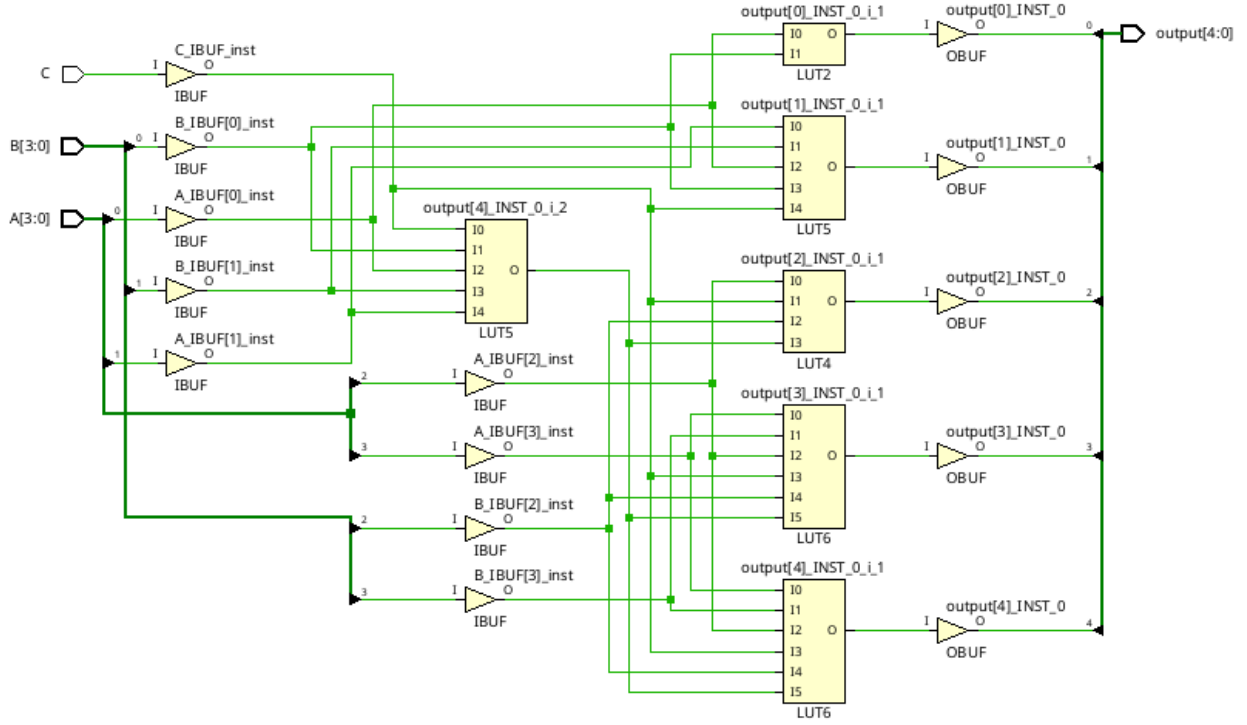


Figure 9: Schematica circuito reale

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	5	0	0	53200	< 0.01
LUT as Logic	5	0	0	53200	< 0.01
LUT as Memory	0	0	0	17400	0.00

Table 2: Numero LUT utilizzati nel circuito a 4 bit

3.2 8 bit

Analizziamo ora le simulazioni ottenute nel caso di circuito a 8 bit.

3.2.1 Behavioural

La *Figure 10* illustra la simulazione behavioural, quindi ideale.

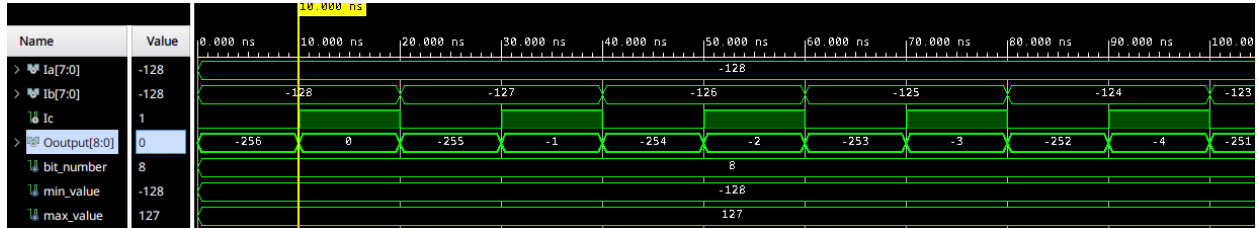


Figure 10: Simulazione 8 bit

3.2.2 Post-implementation

Passando alla simulazione post-implementation, notiamo che il ritardo inizia a diventare piuttosto importante. Dalla *Figure 11* vediamo infatti che per avere il primo valore utile saranno necessari ben 7,5 ns sui 10 totali concessi.

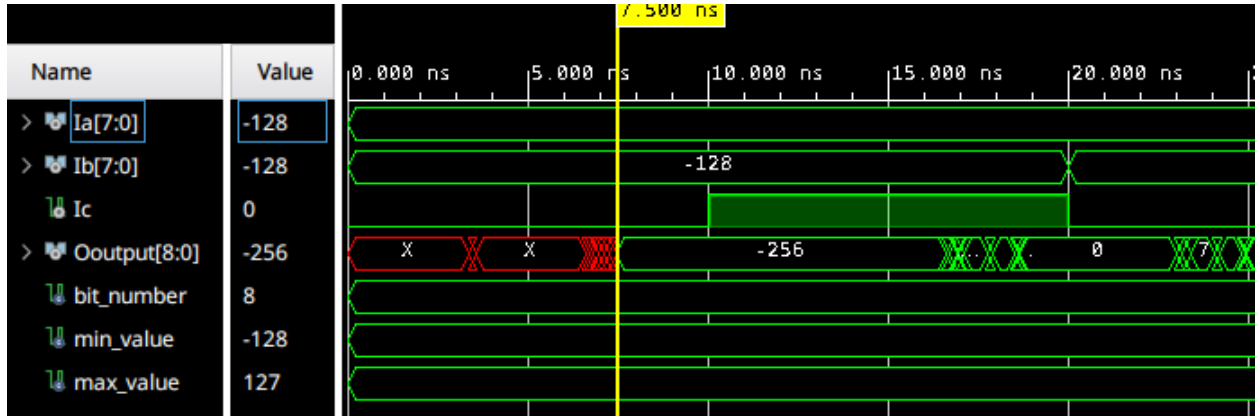


Figure 11: Primo valore utile

Negli istanti successivi il valore del ritardo tende ad assestarsi su una media di circa 7 ns dalla ricezione degli input all'invio dell'output corretto, come visibile dall'esempio generico della *Figure 12*

Dal punto di vista della temperatura il circuito ad 8 bit inizia a risentire della mancanza di un clock. La *Figure 13* infatti mostra una temperatura massima raggiunta di oltre 100 gradi, che rischia di compromettere la stabilità ed il funzionamento di un circuito reale.

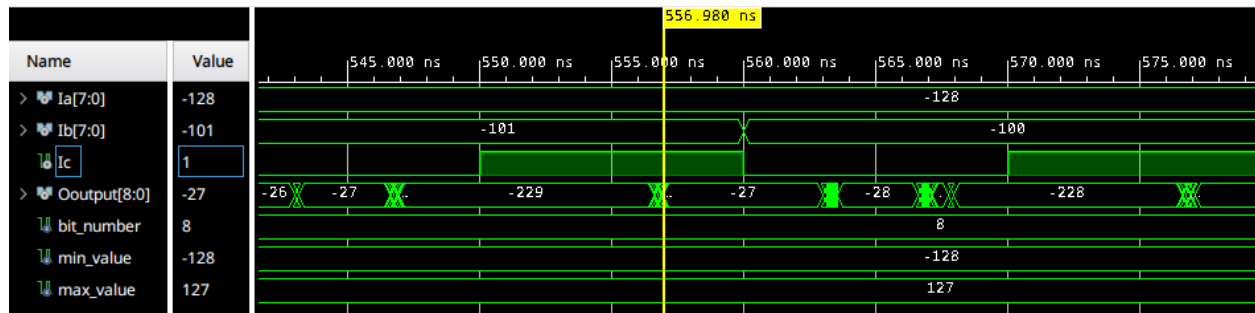


Figure 12: Dimostrazione ritardo di 6,98 ns

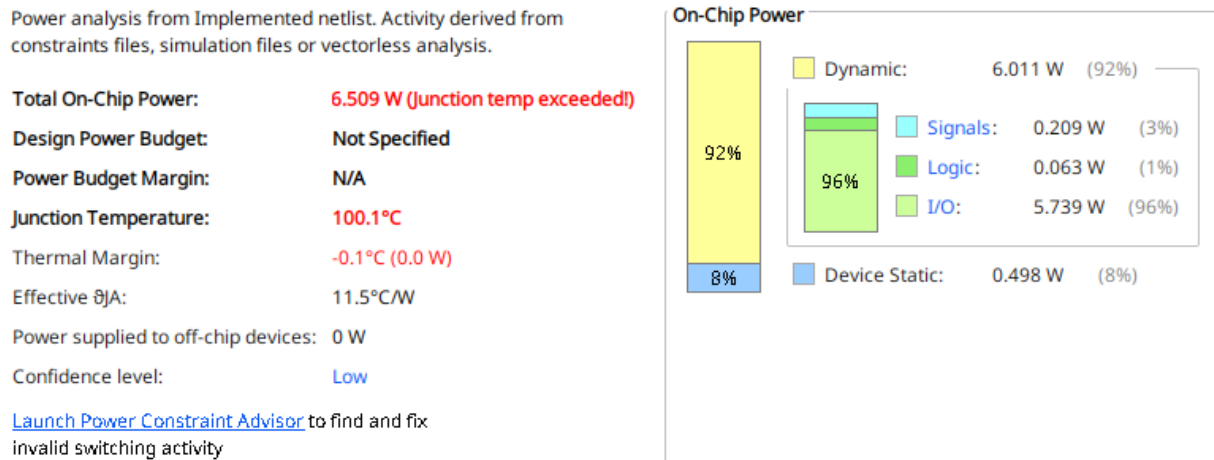


Figure 13: Total On-Chip Power e Junction Temperature

3.2.3 LUT Tables

Le tables utilizzate nel circuito a 8 bit sono maggiori di quelle del circuito a 4 bit, come vediamo dalle quantità mostrate in *Table 3* e dalla loro disposizione, evidenziata in *Figure 14*

Nome	Utilizzate	Tipo di utilizzo
IBUF	17	IO
OBUF	9	IO
LUT6	6	LUT
LUT5	3	LUT
LUT4	2	LUT
LUT2	1	LUT

Table 3: Numero LUT utilizzati nel circuito a 8 bit

Nonostante questi incrementi, l'utilizzazione mostrata in figura *Table 4* rimane minima.

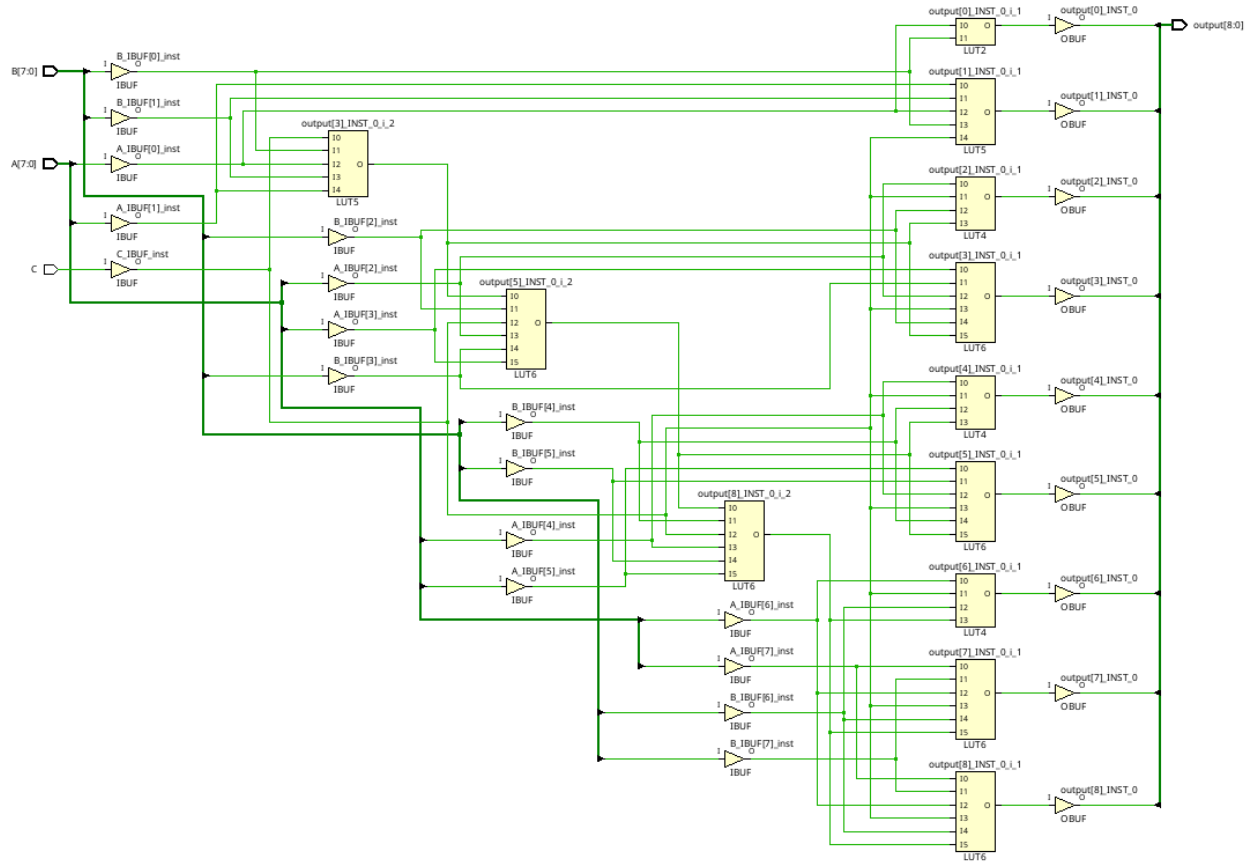


Figure 14: Schematica circuito reale

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	11	0	0	53200	0.02
LUT as Logic	11	0	0	53200	0.02
LUT as Memory	0	0	0	17400	0.00

Table 4: Numero LUT utilizzati nel circuito a 4 bit

3.3 16 Bit