



Master Degree in Embedded Computing Systems
A.Y. 2016 – 2017

Digital System Design

Report of Convolutional Code Generator

Giovanni Falzone
falzone.giovanni2@gmail.com
Student ID: 464756
Delivery date: 14/07/17

Indice

1. Introduzione
 - a. Codici Convoluzionali
 - b. Studiare il comportamento di un Convolutional Code Encoder
2. Descrizione dell'Architettura
3. TestBench
 - a. Test 1, input "111111111111"
 - b. Test 2, input "101010101010"
4. Sintesi con Vivado Tool
 - a. Utilizzazione delle risorse FPGA
 - b. Valutazione del Timing durante la fase di Sintesi
 - c. Power Consumption dopo la fase di Sintesi
5. Implementazione con Vivado Tool
6. Codice VHDL
 - a. FF_D.vhd
 - b. FF_D_SHIFTREG.vhd
 - c. GEN_CONV_CODE.vhd
 - d. RC_GeneratorVect.vhd
7. Simulatore in Python
 - a. Inserimento 8 bit "00000101", I8 5

1 Introduzione

Il processo di codifica di canale consiste nell'aggiungere bit di ridondanza al messaggio che si vuole trasmettere.

In fase di ricezione, la presenza di tali bit consente di rilevare o correggere eventuali errori introdotti nel messaggio dal rumore presente sul canale.

Lo svantaggio della codifica di canale è che si devono trasmettere più bit di quanti ne siano effettivamente necessari per rappresentare il messaggio, aumenta di conseguenza il tempo richiesto per la trasmissione. Per ogni tipo di codice possiamo misurarne le prestazioni come:

- Capacità di Rilevazione, il numero massimo di errori che riesce a rilevare
- Capacità di Correzione, il numero massimo di errori che riesce a correggere
- Rate $R_c = \frac{k}{n}$ ovvero dati k bit di messaggio quanti bit è necessario trasmettere
- Complessità realizzativa dell'architettura

I codici possono essere divisi in due categorie:

I **Codici a Blocchi**, i quali hanno in ingresso un blocco di k -simboli ai quali aggiungono q -bit di ridondanza ottenendo *codeword* di $n = k + q$ simboli, in particolare nei codici a blocchi non c'è correlazione tra i simboli di due diverse *codeword*.

Codici Convolutionali, i quali trasmettono uno stream di simboli ed ogni simbolo influenza la generazione delle *codeword* dei simboli successivi.

1.a Codici Convolutionali

I codici convolutionali vengono utilizzati per ottenere un trasferimento di dati affidabile in applicazioni quali trasferimento di video digitale, la radio, la telefonia mobile e le comunicazioni via satellite.

Constraint length più grandi producono codici più potenti ma anche più complessi da realizzare e richiedono una complessità esponenziale nella fase di decodifica.

Sono utilizzati nelle missioni spaziali per comunicare nello spazio profondo, durante le missioni Voyager lanciate 1977 sono stati utilizzati Codici Convolutionali con constraint length $k = 15$ e rate $R_c = \frac{1}{2}$, per le missioni successive Mars Pathfinder (1996) e Missione spaziale Cassini-Huygens(1997) sono stati adottati Codici Convolutionali $k = 15$ e rate $R_c = \frac{1}{6}$.

Un Convolutional Encoder viene rappresentato dai parametri (n, k, L) o (R_c, L) , e consiste in una Finite State Machine composta da uno shift register di L stadi ognuno dei quali è connesso opportunamente a n sommatore modulo-2, ogni messaggio m_i influenza i successivi $n(L + 1)$, quest'ultima viene definita come *Constraint Length*, definiamo invece come *Memoria* la quantità L che è la dimensione dello shift register.

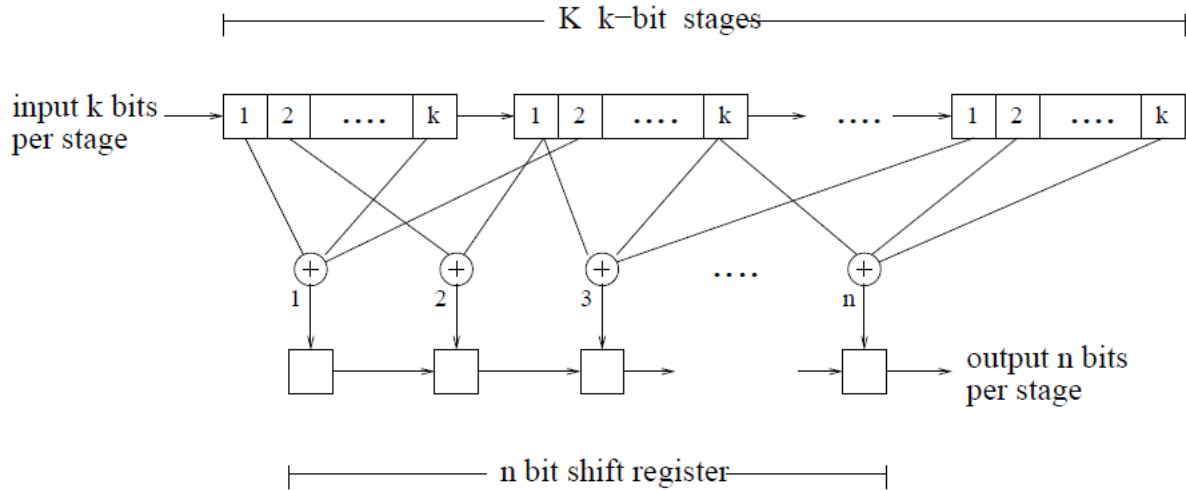
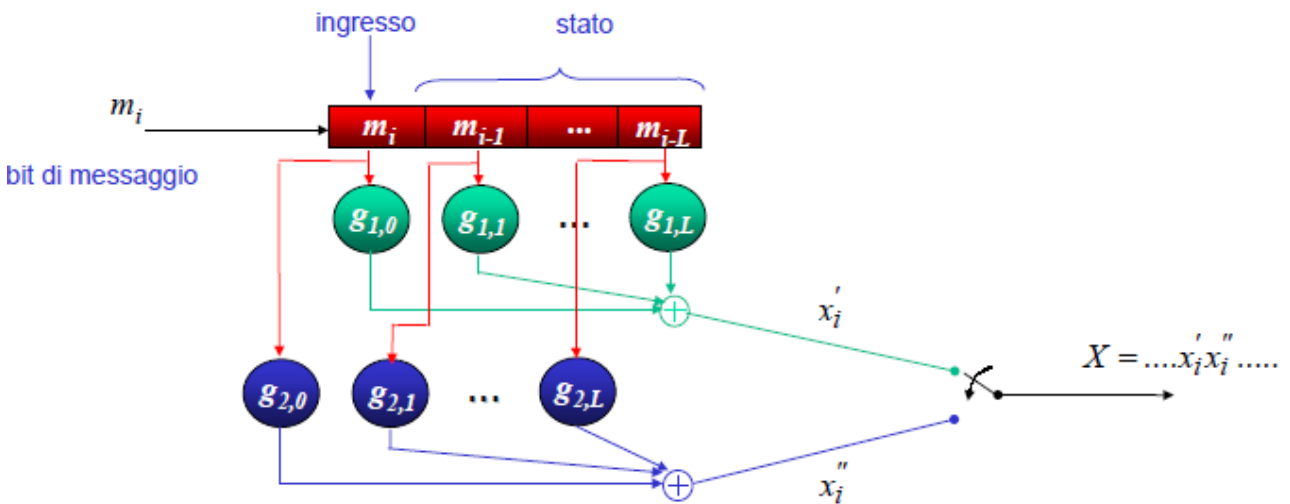


Figure 1, Block diagram of a general convolutional encoder.

Le connessioni tra i vari stage dello shift register e i sommatore modulo-2 vengono espresse da n vettori generatori ognuno di dimensione Kk , uno per ogni sommatore modulo-2, un "1" nella i -esima posizione indica che il corrispondente stage dello shift register è connesso con il sommatore modulo-2, "0" altrimenti.

Il seguente schema corrisponde ad un Codice Convolutionale $(1, 2, L)$, le sequenze g_i sono le sequenze generatrici.



$$x'_i = \sum_{j=0}^L m_{(i-j)} g_{1,j}$$

$$x''_i = \sum_{j=0}^L m_{(i-j)} g_{2,j}$$

1.b Studiare il comportamento di un Convolutional Code Encoder

Il comportamento di un Convolutional Encoder può essere studiato tramite tre diversi metodi:

- Tree Diagram
- Trellis Diagram
- State Diagram

Il **Tree Diagram**, studiamo il comportamento dell'encoder partendo dallo stato iniziale e studiandone l'evoluzione ad ogni possibile input, di conseguenza abbiamo 2^k rami uscenti per ogni nodo e il comportamento si ripete al dopo L livelli con L lunghezza del vincolo.

Il **Trellis Diagram**, partendo dal Tree diagram e considerando tutti i possibili stati che può avere lo shift register notiamo che se un nodo ha lo stesso stato di un altro e per ogni input otteniamo uno stesso output allora possiamo unire questi due nodi in uno solo, applicando questa tecnica per ogni nodo otteniamo una forma più compatta che è il Trellis Diagram.

Lo **State Diagram**, è la forma più compatta ed è composta da un grafo con 2^L nodi, uno per ogni possibile stato ed ognuno di questi con 2^k transizioni uscenti e 2^k entranti.

Considerando un Codice Convolutionale (2,1,2) con $g_1=[1,1,1]$ e $g_2=[1,0,1]$ otteniamo i seguenti diagrammi.

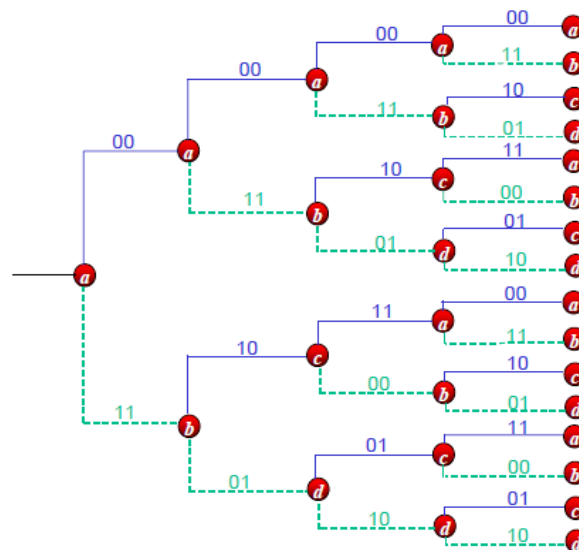


Figura 2, Tree Diagram

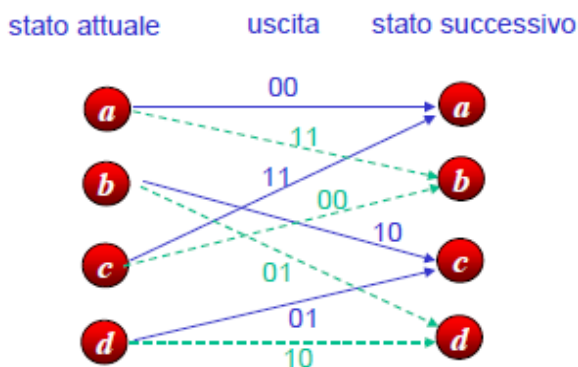


Figura 3, Trellis Diagram

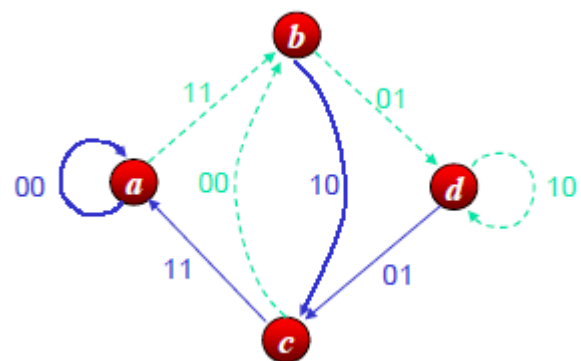
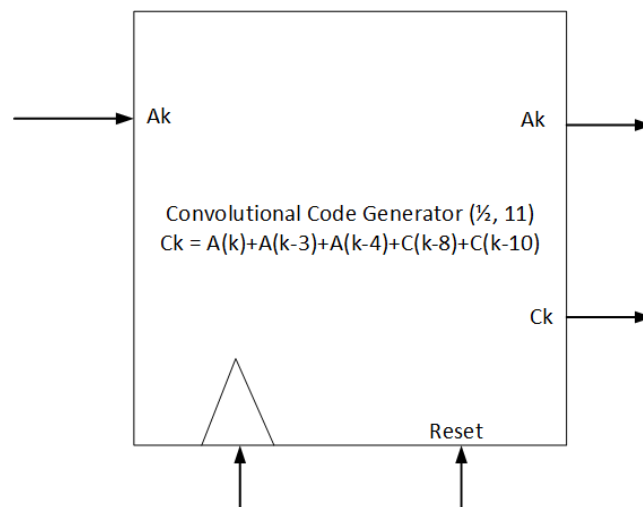


Figura 2, State Diagram

2 Descrizione dell'Architettura

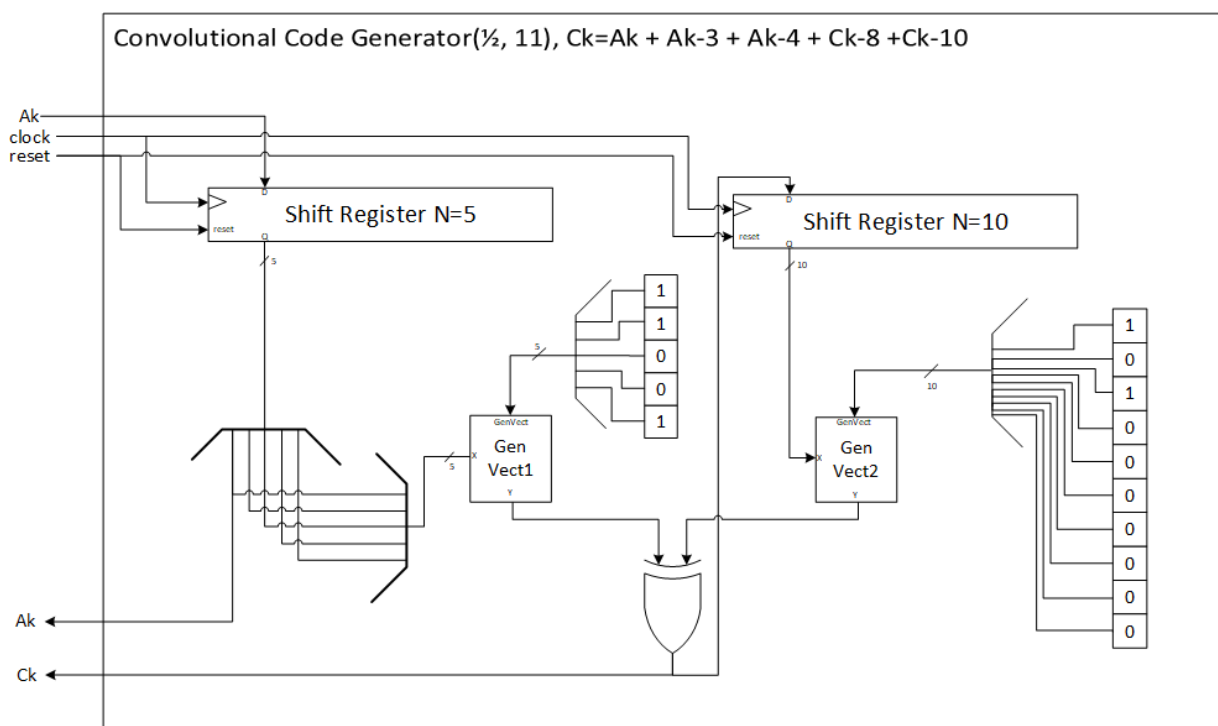
Il sistema è costituito da tre ingressi, bit in ingresso A_k , clock e reset, in uscita ritroviamo A_k che è il medesimo segnale in ingresso ritardato di un clock e il bit C_k relativo all'implementazione della funzione generatrice del codice.

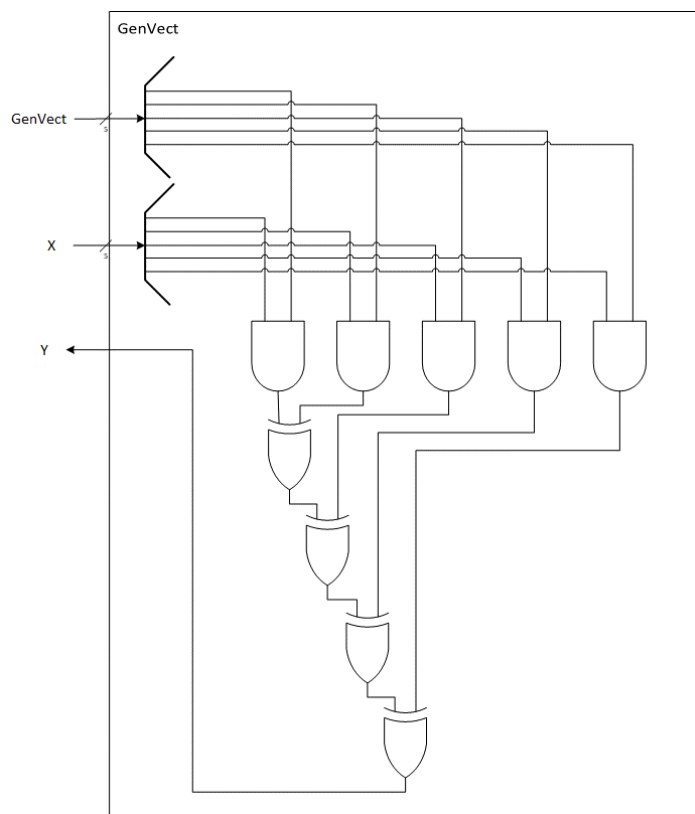


Il Convolutional Code si compone di due **Shift Register** realizzati tramite componenti DFF istanziati tramite una Generate structure, l'uscita di ogni stadio è connessa con l'ingresso dello stadio successivo, fatta eccezione per l'input del primo stadio che costituisce l'ingresso dello Shift Register.

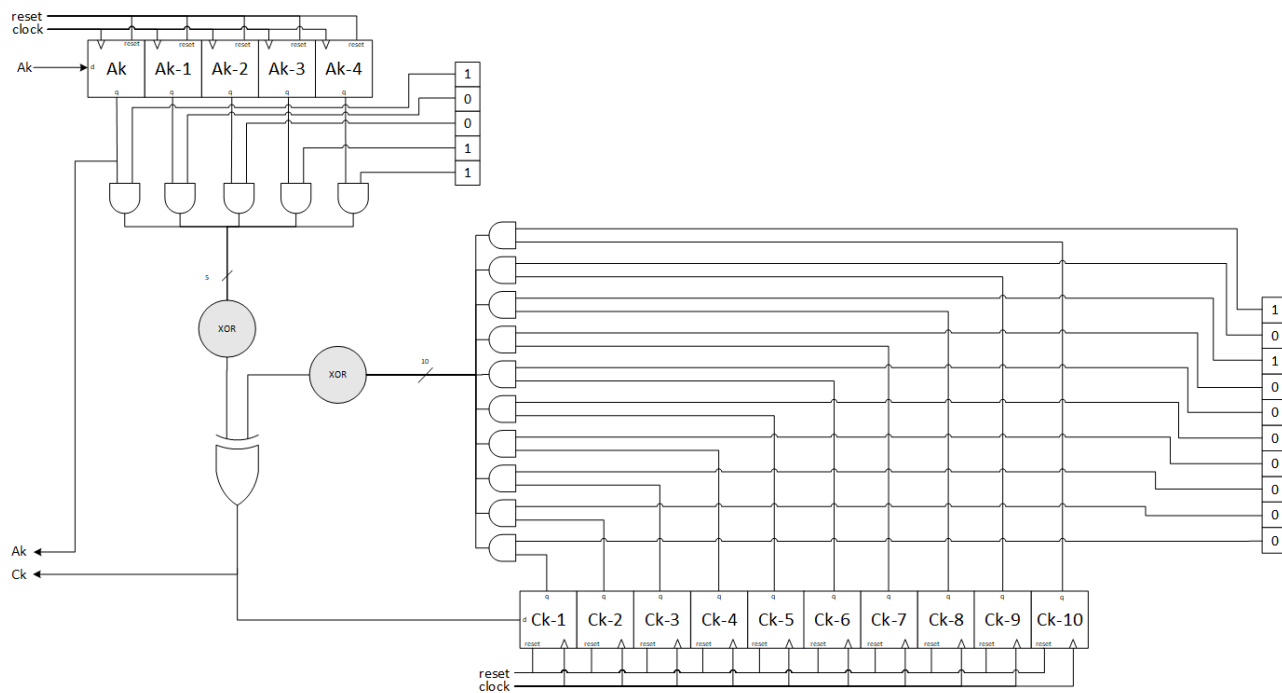
L'output di ogni stadio viene riportato come vettore di bit in uscita dallo Shift Register per poterne valutare lo stato.

Il modulo **GenVect** viene utilizzato congiuntamente allo Shift Register e si occupa di realizzare la logica relativa al Vettore Generatore, ha come input gli output di ogni stadio dello Shift Register ed un vettore di bit ognuno ad indicare la connessione di ogni stadio dello Shift Register.





Il Modulo Gen Vect descritto in GEN_CONV_CODE.vhd implementa il vettore generatore, ha come ingresso lo stato dello Shift Register x e la sequenza generatrice *GenVect*.



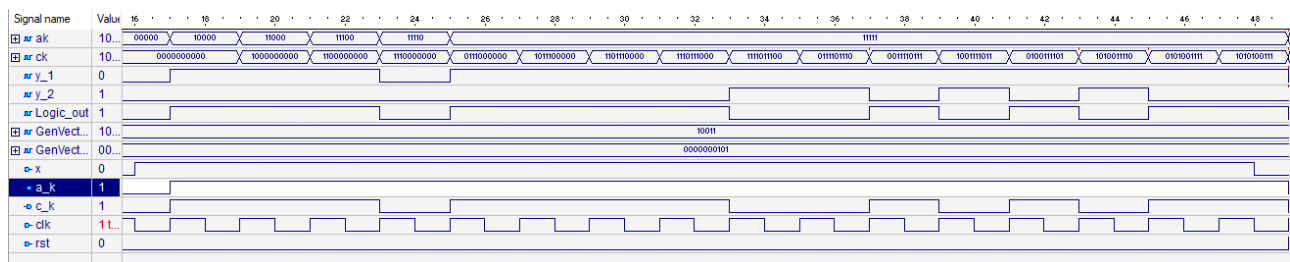
3 Testbench

Per testare il generatore di Codici Convoluzionali verifico che il modulo raggiunga tutti gli stati previsti e quindi generi il corretto output per ogni input in ingresso.

Al fine di testare il modulo implementato in VHDL, ho realizzato un tool in Python che mi permetta di eseguire il medesimo algoritmo e di confrontarne lo stato del modulo in fase di simulazione, il tool permette di testare l'inserimento di un singolo bit o di uno stream di bit espresso come numero intero, permette inoltre di salvare l'evoluzione dello stato del modulo in formato csv in modo da analizzarne l'evoluzione come nelle seguenti tabelle.

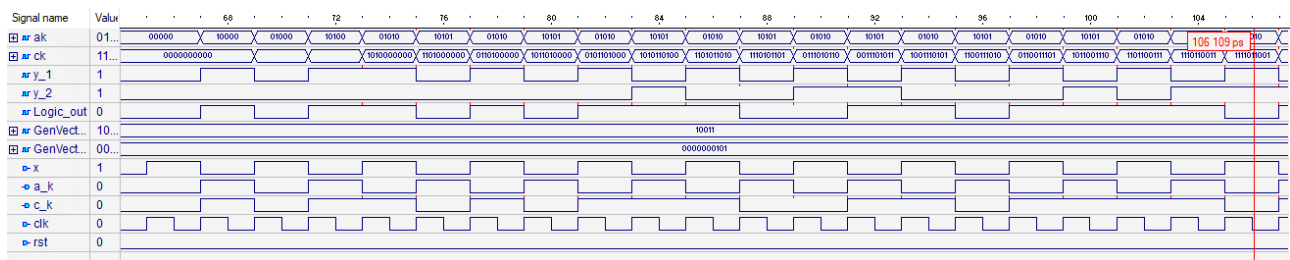
3.a Test 1, input "111111111111"

#	Input	ShiftRegA	ShiftRegC	Ak	Ck
1	1	10000	0000000000	1	1
2	1	11000	1000000000	1	1
3	1	11100	1100000000	1	1
4	1	11110	1110000000	1	0
5	1	11111	0111000000	1	1
6	1	11111	1011100000	1	1
7	1	11111	1101110000	1	1
8	1	11111	1110111000	1	1
9	1	11111	1111011100	1	0
10	1	11111	0111101110	1	0
11	1	11111	0011110111	1	1
12	1	11111	1001111011	1	0
13	1	11111	0100111101	1	1
14	1	11111	1010011110	1	0
15	1	11111	0101001111	1	1
16	1	11111	1010100111	1	1



3.b Test 2, input “101010101010”

#	Input	ShiftRegA	ShiftRegC	Ak	Ck
1	1	10000	0000000000	1	1
2	0	01000	1000000000	0	0
3	1	10100	0100000000	1	1
4	0	01010	1010000000	0	1
5	1	10101	1101000000	1	0
6	0	01010	0110100000	0	1
7	1	10101	1011010000	1	0
8	0	01010	0101101000	0	1
9	1	10101	1010110100	1	1
10	0	01010	1101011010	0	1
11	1	10101	1110101101	1	0
12	0	01010	0111010110	0	0
13	1	10101	0011101011	1	1
14	0	01010	1001110101	0	1
15	1	10101	1100111010	1	0
16	0	01010	0110011101	0	1



4 Sintesi con Vivado Tool

4.a Utilizzazione delle risorse FPGA

Il tool Vivado in seguito alla fase di Sintesi ci mostra le risorse FPGA utilizzate.

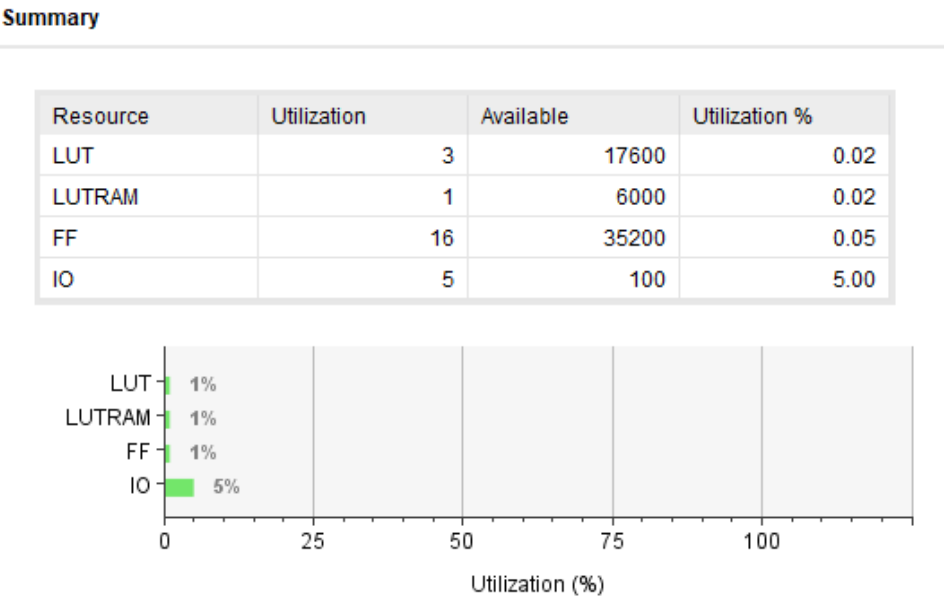


Figura 3, Utilization

4.b Valutazione del Timing durante la fase di Sintesi

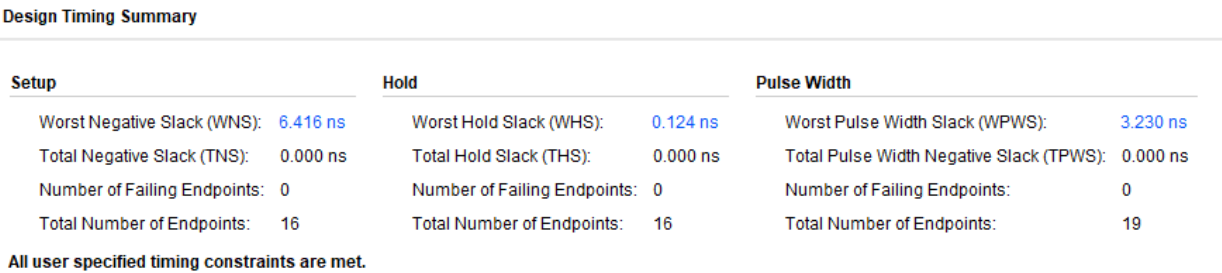


Figura 4, Timing Summary con Period 8ns

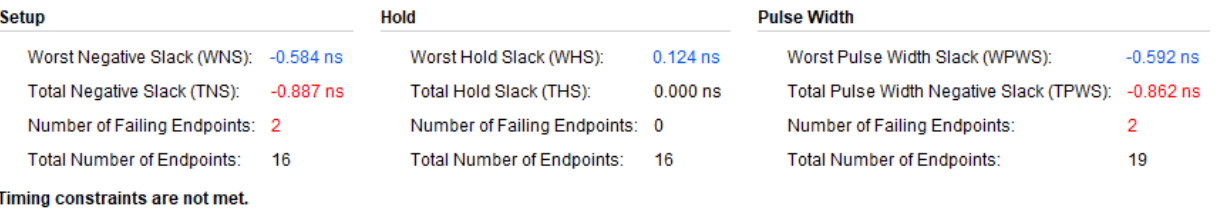


Figura 5, Timing Summary con Period 1ns

Il tool Vivado permette di analizzare il percorso critico e di valutare il timing richiesto, Il tempo necessario per il percorso critico è: 1.592ns e dunque la frequenza massima 628MHz.

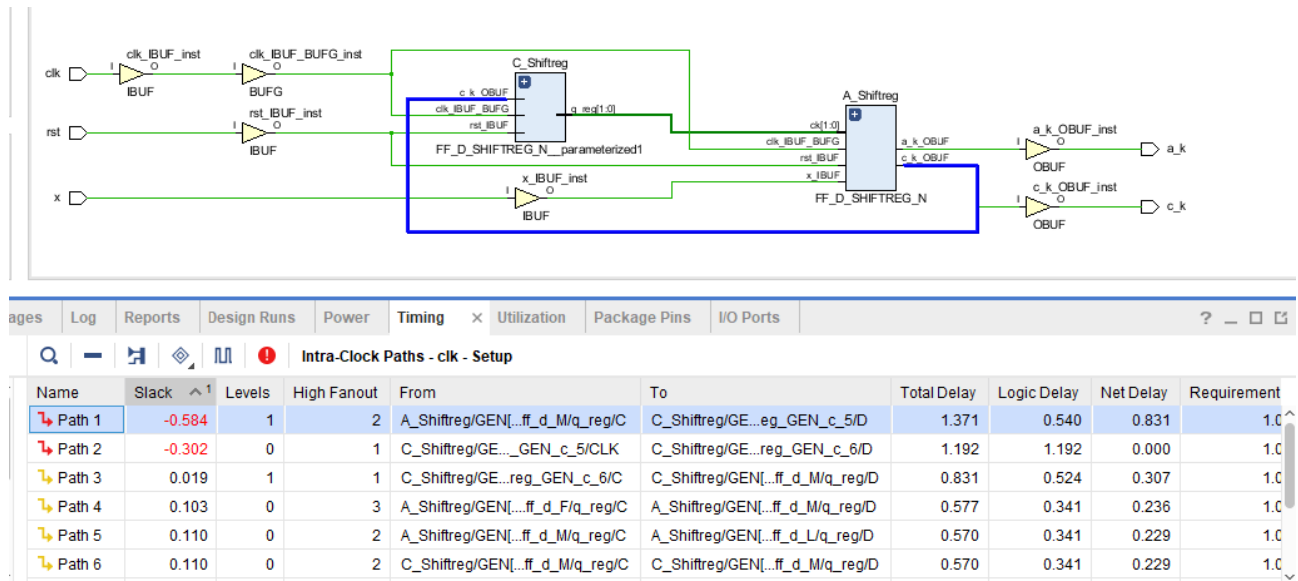


Figura 6, Critical Path

Vado a settare come Timing constraint del clock proprio questo periodo minimo richiesto ed i vincoli temporali sono rispettati con la massima frequenza.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.008 ns	Worst Hold Slack (WHS): 0.124 ns	Worst Pulse Width Slack (WPWS): 0.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 16	Total Number of Endpoints: 16	Total Number of Endpoints: 19

All user specified timing constraints are met.

Figura 7, Timing Summary con Period 1.592ns

4.c Power Consumption dopo la fase di Sintesi

In seguito alla fase di sintesi possiamo valutare il Power Consumption previsto per l'architettura realizzata utilizzando come parametri gli standard domestici, il power consumption totale è di 0.116W di cui il 12% relativo alla Potenza Dinamica e l'88% alla Potenza statica.

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power: 0.116 W
Junction Temperature: 26.3 °C
Thermal Margin: 58.7 °C (5.0 W)
Effective θ_{JA} : 11.5 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Medium

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

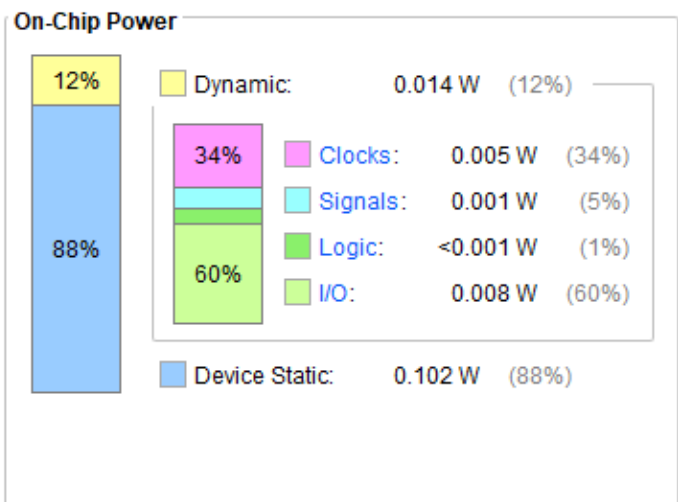


Figura 8, Power Summary

5 Implementazione con Vivado Tool

Durante la fase di Implementazione Vivado prova a implementare la sintesi nella board Zynq-7000 FPGA utilizzando i constraints precedentemente definiti ed otteniamo i seguenti risultati per quanto riguarda Utilizzazione, Timing e Power Consumption.

Summary

Resource	Utilization	Available	Utilization %
LUT	3	17600	0.02
LUTRAM	1	6000	0.02
FF	17	35200	0.05
IO	5	100	5.00

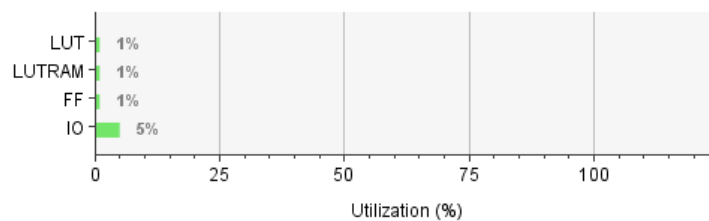


Figura 10, Utilization

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.342 ns	Worst Hold Slack (WHS): 0.108 ns	Worst Pulse Width Slack (WPWS): 0.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 15	Total Number of Endpoints: 15	Total Number of Endpoints: 19

All user specified timing constraints are met.

Figura 9, Timing Summary con Period 1.592ns

Summary

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

Total On-Chip Power: 0.113 W
Junction Temperature: 26.3 °C
 Thermal Margin: 58.7 °C (5.0 W)
 Effective θ_{JA} : 11.5 °C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

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

On-Chip Power

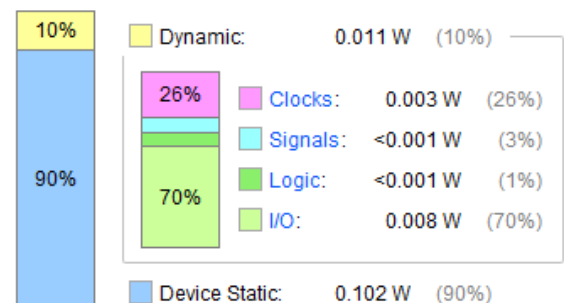


Figura 11, Power Summary

6 Codice VHDL

6.a FF_D.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity FF_D is
5      port (
6          d      : in std_ulogic;
7          q      : out std_ulogic;
8          clk    : in std_ulogic;
9          rst    : in std_ulogic
10     );
11 end FF_D;
12
13 architecture rtl of FF_D is
14 begin
15     FF_D_p : process(clk,rst)
16     begin
17         if rst = '1' then
18             q <= '0';
19         elsif (clk = '1' and clk'event) then
20             q <= d;
21         end if;
22     end process;
23 end rtl;
```

6.b FF_D_SHIFTREG.vhd

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity FF_D_SHIFTREG_N is
5      generic (Nbit : positive := 8);
6      port(
7          d : in std_ulogic;
8          q : out std_ulogic_vector(0 to Nbit - 1);  -- output dello stato
9          clk : in std_ulogic;
10         rst : in std_ulogic
11     );
12 end FF_D_SHIFTREG_N;
13
14 architecture rtl of FF_D_SHIFTREG_N is
15     component FF_D
16     port (
17         d : in std_ulogic;
18         q : out std_ulogic;
19         clk : in std_ulogic;
20         rst : in std_ulogic
21     );
22     end component FF_D;
23
24     signal wire : std_ulogic_vector(0 to Nbit - 2);
25
26 begin
27     -- generation il primo FFD Ã" connesso all'ingresso, gli altri hanno come ingresso
28     -- l'output dello stadio precedente, l'output di ogni stadio Ã" riportato come output del modulo
29     GEN: for i in 0 to Nbit-1 generate
30         FIRST: if i = 0 generate
31             ff_d_F : FF_D port map(d, wire(i), clk, rst);
32             q(i) <= wire(i);
33         end generate FIRST;
34
35         MIDDLE: if i > 0 and i < Nbit-1 generate
36             ff_d_M : FF_D port map(wire(i-1), wire(i), clk, rst);
37             q(i) <= wire(i);
38         end generate MIDDLE;
39
40         LAST: if i = Nbit-1 generate
41             ff_d_L : FF_D port map(wire(i-1), q(i), clk, rst);
42         end generate LAST;
43     end generate GEN;
44 end rtl;

```

6.c GEN_CONV_CODE.vhd

```

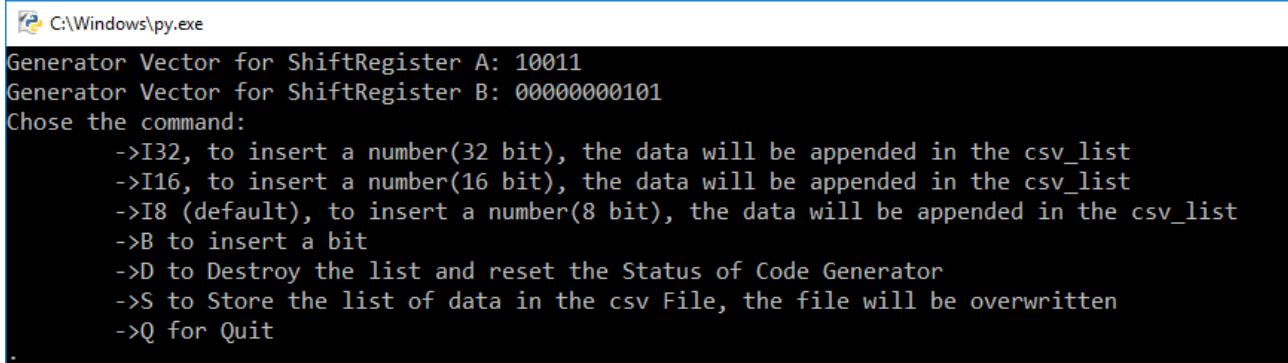
11 entity GEN_CONV_CODE is
12     port (
13         x : in std_ulogic;
14         a_k : out std_ulogic;
15         c_k : out std_ulogic;
16         clk : in std_ulogic;
17         rst : in std_ulogic
18     );
19 end GEN_CONV_CODE;
20
21 architecture rtl of GEN_CONV_CODE is
22     component FF_D_SHIFTREG_N
23         generic (Nbit : positive := 8);
24         port(
25             d : in std_ulogic;
26             q : out std_ulogic_vector(0 to Nbit - 1);
27             clk : in std_ulogic;
28             rst : in std_ulogic
29         );
30     end component FF_D_SHIFTREG_N;
31
32     component RC_GeneratorVect
33         generic (Nbit : positive := 16);
34         port (
35             GenVect : in std_ulogic_vector(0 to Nbit-1);
36             x : in std_ulogic_vector(0 to Nbit-1);
37             y : out std_ulogic
38         );
39     end component RC_GeneratorVect;
40
41     constant Nbit : positive := 5;          -- dimensione primo ShiftRegister A
42     constant GenVect_1 : positive := (16+2+1); -- costante vettore 10011 per A
43
44     constant Mbit : positive := 10;         -- dimensione secondo Shift Register B
45     constant GenVect_2 : positive := (4+1);  -- costante vettore 000000101 per B
46
47     signal ak : std_ulogic_vector(0 to Nbit-1); -- wire in uscita da ogni stadio dello ShiftRegister A
48     signal ck : std_ulogic_vector(0 to Mbit-1); -- wire in uscita da ogni stadio dello ShiftRegister B
49
50     signal y_1 : std_ulogic; -- wire in uscita dal GenVect1
51     signal y_2 : std_ulogic; -- wire in uscita dal GenVect2
52
53     signal Logic_out : std_ulogic; -- wire in ingresso al secondo ShiftRegister C
54
55     signal GenVectSignal1 : std_ulogic_vector(0 to Nbit-1); -- wire in ingresso al GenVect1
56     signal GenVectSignal2 : std_ulogic_vector(0 to Mbit-1); -- wire in ingresso al GenVect2
57
58 begin
59     GenVectSignal1 <= std_ulogic_vector(TO_UNSIGNED(GenVect_1, Nbit)); -- conversione ed assegnamento del valore
60     GenVectSignal2 <= std_ulogic_vector(TO_UNSIGNED(GenVect_2, Mbit)); -- dei due Vettori Generatori
61
62     Logic_out <= y_1 xor y_2;
63     c_k <= Logic_out; -- output Ck
64     a_k <= ak(0); -- output Ak
65
66     A_Shiftreg : FF_D_SHIFTREG_N
67         generic map(Nbit => Nbit)
68         port map(
69             d => x,
70             q => ak,
71             clk => clk,
72             rst => rst
73         );
74
75     GenVect1 : RC_GeneratorVect
76         generic map(Nbit => Nbit)
77         port map(
78             GenVect => GenVectSignal1,
79             x => ak,
80             y => y_1
81         );
82
83     GenVect2 : RC_GeneratorVect
84         generic map(Nbit => Mbit)
85         port map(
86             GenVect => GenVectSignal2,
87             x => ck,
88             y => y_2
89         );
90
91     C_Shiftreg : FF_D_SHIFTREG_N
92         generic map(Nbit => Mbit)
93         port map(
94             d => Logic_out,
95             q => ck,
96             clk => clk,
97             rst => rst
98         );
99 end rtl;
100

```


6.d RC_GeneratorVect.vhd

```
9  entity RC_GeneratorVect is
10      generic (Nbit : positive :=16);
11      port (
12          GenVect : in std_ulogic_vector(0 to Nbit-1);    -- input Vettore Generatore
13          x      : in std_ulogic_vector(0 to Nbit-1);    -- input Settoze Stato dello ShiftRegister
14          y      : out std_ulogic                        -- output
15      );
16  end RC_GeneratorVect;
17
18  architecture rtl of RC_GeneratorVect is
19      signal a : std_ulogic_vector(0 to Nbit-1);    -- wire uscita della AND tra Stato e Vettore Generatore
20      signal par : std_ulogic_vector(0 to Nbit-2);  -- wire per realizzare la cascata di XOR
21
22      begin
23          a <= x and GenVect;
24
25      -- generation
26      GEN: for i in 0 to Nbit-1 generate
27          FIRST: if i = 0 generate
28              par(0) <= a(0) xor a(1);    -- primo xor necessita dei primi due ingressi
29          end generate FIRST;
30
31          MIDDLE: if i > 0 and i < Nbit-1 generate
32              par(i) <= par(i-1) xor a(i); -- cascata di XOR intermedi
33          end generate MIDDLE;
34
35          LAST: if i = Nbit-1 generate
36              y <= par(i-1) xor a(i);    -- output dell'ultimo XOR  $\bar{A}$  anche l'output del modulo
37          end generate LAST;
38      end generate GEN;
39  end rtl;
```

7 Simulatore in Python



```
C:\Windows\py.exe
Generator Vector for ShiftRegister A: 10011
Generator Vector for ShiftRegister B: 0000000101
Chose the command:
->I32, to insert a number(32 bit), the data will be appended in the csv_list
->I16, to insert a number(16 bit), the data will be appended in the csv_list
->I8 (default), to insert a number(8 bit), the data will be appended in the csv_list
->B to insert a bit
->D to Destroy the list and reset the Status of Code Generator
->S to Store the list of data in the csv File, the file will be overwritten
->Q for Quit
.
```

Lo script permette di inserire un bit per volta con il comando *B* o un intero con il comando *I* su 8/16/32 bit, per quest'ultima opzione viene tenuta in considerazione la rappresentazione in base 2 dell'intero e viene eseguito l'algoritmo per gli 8/16/32 bit consecutivi della rappresentazione inserendo a partire dal LSB.

Ad ogni inserimento di un bit viene salvato lo stato del modulo in un'apposita lista che è possibile salvare in formato csv tramite il comando *S*.

È possibile resettare il modulo e distruggere la lista con il comando *D*, in questo caso lo stato dei registri viene riportato a 0.

7.a Inserimento 8 bit "00000101", l8 5

```
C:\Windows\py.exe
Generator Vector for ShiftRegister A: 10011
Generator Vector for ShiftRegister B: 0000000101
Chose the command:
->I32, to insert a number(32 bit), the data will be appended in the csv_list
->I16, to insert a number(16 bit), the data will be appended in the csv_list
->I8 (default), to insert a number(8 bit), the data will be appended in the csv_list
->B to insert a bit
->D to Destroy the list and reset the Status of Code Generator
->S to Store the list of data in the csv File, the file will be overwritten
->Q for Quit

:i8
Insert a number:
5
Input Stream: 00000101
-----
x: 1
Shift Register A: 10000
Shift Register C: 000000000
a_k: 1
c_k: 1
-----
x: 0
Shift Register A: 01000
Shift Register C: 100000000
a_k: 0
c_k: 0
-----
x: 1
Shift Register A: 10100
Shift Register C: 010000000
a_k: 1
c_k: 1
-----
x: 0
Shift Register A: 01010
Shift Register C: 101000000
a_k: 0
c_k: 1
-----
x: 0
Shift Register A: 00101
Shift Register C: 110100000
a_k: 0
c_k: 1
-----
x: 0
Shift Register A: 00010
Shift Register C: 111010000
a_k: 0
c_k: 1
-----
x: 0
Shift Register A: 00001
Shift Register C: 111101000
a_k: 0
c_k: 1
-----
x: 0
Shift Register A: 00000
Shift Register C: 111110100
a_k: 0
c_k: 0
-----
Chose the command:
->I32, to insert a number(32 bit), the data will be appended in the csv_list
->I16, to insert a number(16 bit), the data will be appended in the csv_list
->I8 (default), to insert a number(8 bit), the data will be appended in the csv_list
->B to insert a bit
->D to Destroy the list and reset the Status of Code Generator
->S to Store the list of data in the csv File, the file will be overwritten
->Q for Quit
```