



**POLITECNICO
DI TORINO**

*Corso di Laurea Magistrale
in
Ingegneria Elettronica*

Anno Accademico 2017/2018

Sistemi Digitali Integrati

*Relazione di laboratorio
Progetto di un analizzatore di
stati logici*

Studente: Barezzi Mattia

Matricola: 252967

Studente: Borello Gabriele

Matricola: 252845

Studente: Canzonieri Gianmarco

Matricola: 244123

Introduzione

Lo scopo di questo progetto è la realizzazione di un analizzatore di stati logici che, oltre a campionare i segnali su 8 canali, rilevi eventuali glitch, salvi i campioni in una memoria sRam dell'FPGA e sia configurabile, mediante pc, da utente.

Nella relazione di laboratorio sono presenti i seguenti paragrafi:

- Libretto di istruzioni

E successivamente la descrizione dei singoli blocchi

- Top-level
- Sampler
 - Glitch detector
 - Prescaler
- Memory interface
 - Addresser write
 - Addressr read
 - Write_SRAM
 - Read_SRAM
 - Control_SRAM
- PC interface
 - ASCII translator
 - ASCII trasmission
 - UART transmitter
 - UART receiver
- Trigger generator

Per ognuno di essi possiamo trovare

- Pseudocodice
- Datapath
- ASM chart
- Control ASM chart
- Timing
- Simulazioni
- Codice VHDL

Libretto di istruzioni

Collegare per mezzo della linea Rs 232 l'FPGA al computer.

Per un funzionamento corretto della macchina, per prima cosa premere il pulsante Resetn. Successivamente premere il tasto Start.

Tramite tastiera digitare nell'ordine:

- FX' (F maiuscolo e X numero compreso tra 0 e 9): comando per impostare il prescaler della frequenza di campionamento. Sullo schermo comparirà la lettera 'O' se l'operazione è andata a buon fine. In caso venga correttamente inviato il dato a pc o X non sia un valore valido verrà stampata la lettera K.
Nota: Nel caso si voglia lasciare la frequenza di campionamento a 10 MHz si può anche scegliere di non inserire il comando FX.
- TY'Z' (T maiuscolo, Y e Z numeri in esadecimale): comando per impostare l'istante di trigger. Sullo schermo comparirà la lettera 'O' se l'operazione è andata a buon fine. In caso venga inserito un dato non valido o X non sia un valore valido verrà stampata la lettera K.
- S (S maiuscolo): start della macchina. Aspettare finché non è attivo il segnale all_done. A questo punto sono stati memorizzati 128'000 istanti prima del trigger e 128'000 istanti dopo l'evento di trigger.
Nota: non digitare S senza aver digitato T con un valore corretto. In caso venga digitato S prima di aver impostato T la macchina darà un segnale di errore.
- R (R maiuscolo): digitando R vengono stampati a video, uno sotto l'altro, tutti i 256'000 campioni acquisiti. In caso sia presente un glitch all'interno della stringa da 8 bit viene segnalato con una X.
Attendere il segnale attivo di All_done della macchina.

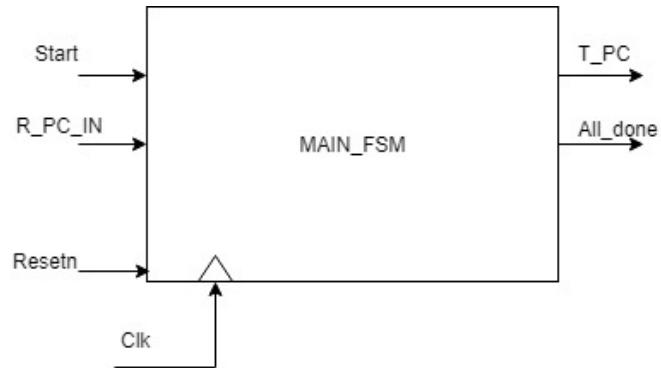
Top-level logic analyzer

La main fsm è divisa in 4 blocchi principali : PC_INTERFACE, TRIGGER , SAMPLE_8 , MEMORY_INTERFACE. Ognuno di questi blocchi nel seguito è descritto nel dettaglio.

Il segnale di resetN resetta tutta la macchina , il segnale di start la avvia. La macchina comunica tramite RS-232 con il PC che invia e riceve segnali dalla board. All_done è un segnale di fine esecuzione.

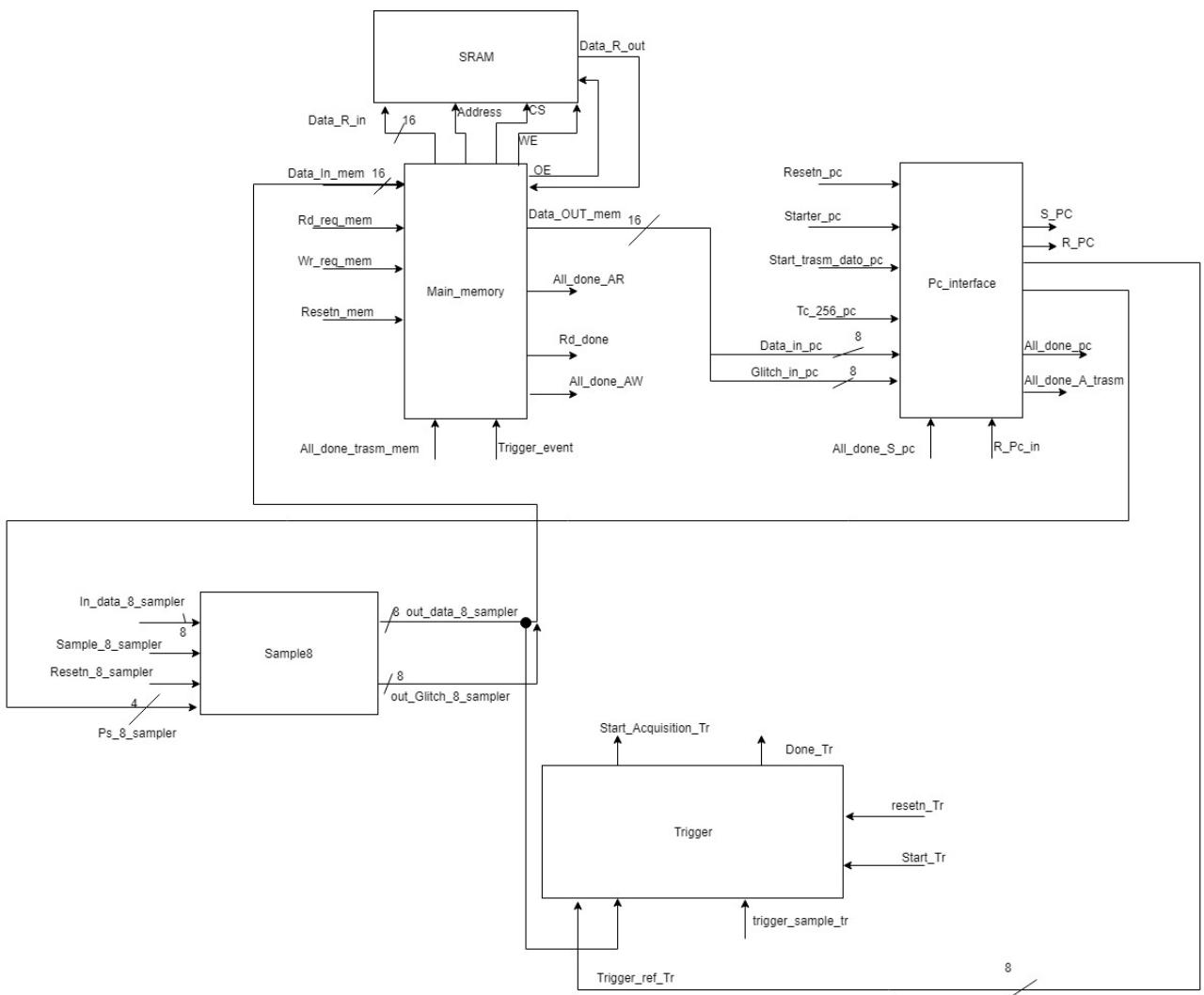
La macchina dopo il segnale di START inizia a campionare a frequenza massima (prescaler settato a 0) e a scrivere di continuo nella sRAM (stato di idle). In questo stato viene impostato F e T. Dopo l'arrivo del segnale S_PC la macchina oltre a scrivere all'interno della memoria cerca l'evento di trigger. Una volta trovato campiona ancora 128000 dati e fornisce infine un ALL_DONE.

Si attende ora il segnale R_PC. Una volta ricevuto verrà stampato a video l'intero contenuto della SRAM 128000 dati prima del trigger e 128000 dati dopo il trigger.



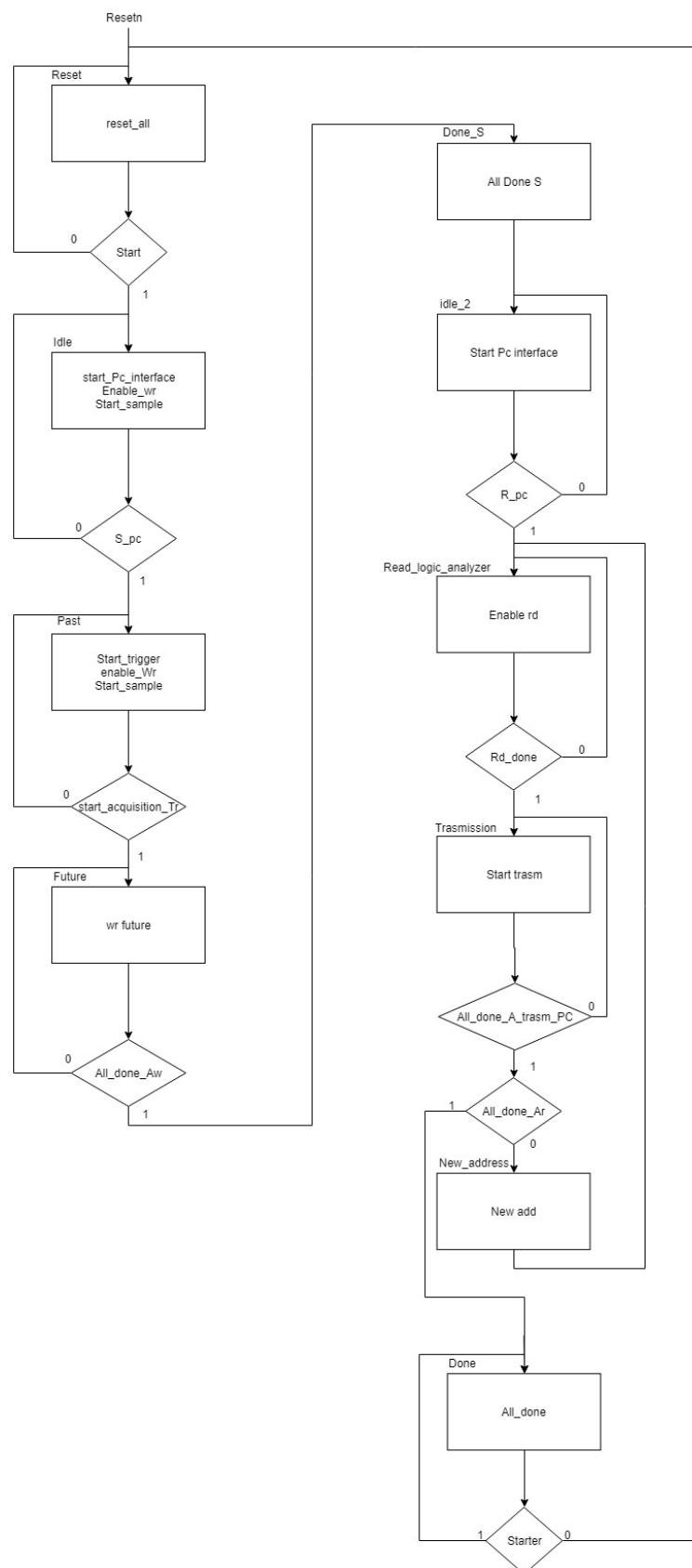
Blocco dell'analizzatore di stati logici

Datapath



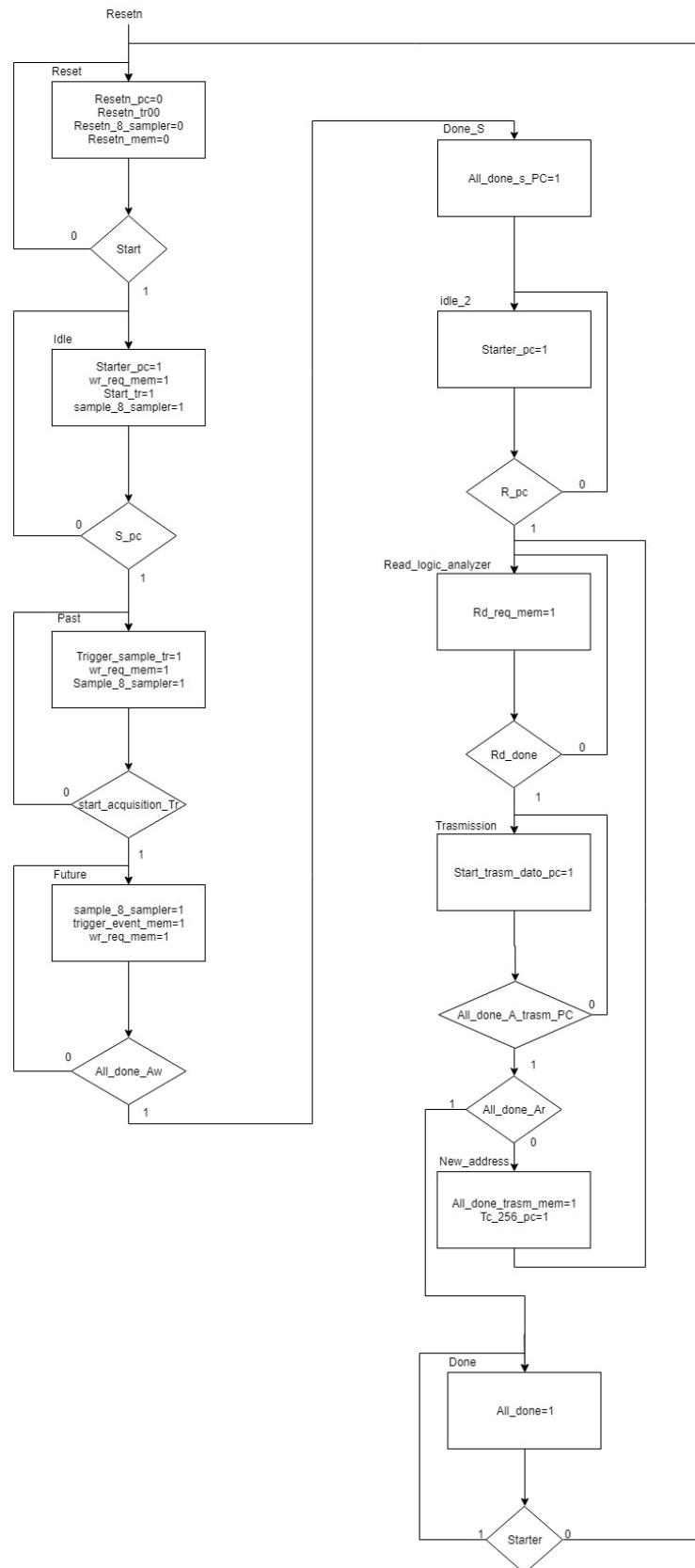
Datapath dell'analizzatore di stati logici

ASM chart



ASM chart dell'analizzatore di stati logici

Control ASM chart



ASM chart dell'analizzatore di stati logici

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: Logic_analyzer.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity logic_analyzer is

port ( start, R_PC , RESETn , clock : in std_logic;
       T_PC , ALL_DONE : OUT STD_LOGIC) ;

END LOGIC_ANALYZER;

ARCHITECTURE BEHAVIOUR OF LOGIC_ANALYZER IS

--SIGNAL

SIGNAL DATA_IN_MEM , DATA_IN_OUT_SRAM : STD_LOGIC_VECTOR (15 DOWNTO 0);

SIGNAL ADDRESS : STD_LOGIC_VECTOR( 17 DOWNTO 0);

SIGNAL PS_8_SAMPLER , F_PC : STD_LOGIC_VECTOR ( 3 DOWNTO 0);

SIGNAL GLITCH_IN_PC , TR, TRIGGER_REFERENCE, SAMPLES, OUT_DATA_8_SAMPLER,
OUT_GLITCH_8_SAMPLER, DATA_OUT_MEM , DATA_IN_pC_INTERFACE ,
DATA_OUT_MEM , IN_DATA_8_SAMPLER : STD_LOGIC_VECTOR ( 7 DOWNTO 0);

signal s_pc, OUT_T, ALL_DONE_A_TRASMISSIONE , ALL_DONE, r_pc, STARTER_PC,
RESETN_PC, TC256_PC, ALL_DONE_S_PC, R_PC_IN, START_TRASM_DATO_PC,
START_ACQUISITION , DONE_T , TRIGGER_SAMPLE , RESETN_TRIGGER,
SAMPLE_8_SAMPLER, RESETN_8_SAMPLER , RD_REQ_MEM , WR_REQ_MEM ,
TRIGGER_EVENT , ALL_DONE_TRASM_MEM , CS_SRAM, WE_SRAM, OE_SRAM ,
ALL_DONE_AW , ALL_DONE_AR , WR_DONE , RD_DONE : STD_LOGIC;
```

--COMPONENT

--MEMORY_INTERFACE

```

COMPONENT Memory_interface IS
PORT(
    --input
    Data_and_glitch: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Trigger_event: IN STD_LOGIC;
    WR_REQ: IN STD_LOGIC;
    RD_REQ: IN STD_LOGIC;
    All_done_trasm: IN STD_LOGIC;
    Clock, Resetn: IN STD_LOGIC;

    --input and output
    --Data_SRAM: INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    DATA_IN_OUT_SRAM: BUFFER STD_LOGIC_VECTOR(15
DOWNTO 0);

    --output
    Add_SRAM: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
    CS_SRAM, WE_SRAM, OE_SRAM: OUT STD_LOGIC;
    Data_IN_PC_Interface: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    ALL_DONE_AW, ALL_DONE_AR: OUT STD_LOGIC;
    WR_DONE, RD_DONE: OUT STD_LOGIC
);

END COMPONENT;

```

-TRIGGER

```

COMPONENT trigger is

Port ( trigger_reference : IN STD_LOGIC_VECTOR (7 downto 0);
samples :IN STD_LOGIC_VECTOR (7 downto 0);
trigger_sample, starto_trigger, Clock : in STD_LOGIC;
RSTn_TRIGGER : IN STD_LOGIC;
start_acquisition : OUT std_logic; -- BUFFER: both IN & OUT
done : OUT STD_LOGIC); --forse in teoria è quello di prima

end COMPONENT;

```

-SAMPLE8

```
COMPONENT Sample8 IS
```

```

PORT( clock: in std_logic;
in_data_8: in std_logic_vector(7 downto 0);
sample_8: in std_logic;
resetn_8: in std_logic;

```

```

ps_8: in std_logic_vector(3 downto 0);
out_data_8: out std_logic_vector(7 downto 0);
out_glitch_8:out std_logic_vector(7 downto 0));
END COMPONENT;

```

```

COMPONENT Pc_interface IS
PORT(      Starter: IN STD_LOGIC;
Clock,Resetn: IN STD_LOGIC;
Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
TC512: IN STD_LOGIC;
All_done_s: IN STD_LOGIC;
R_Pc: IN STD_LOGIC;
Start_trasm_dato: IN STD_LOGIC;
glitch_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
S: OUT STD_LOGIC;
R: OUT STD_LOGIC;
Tr: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
F: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
out_T: OUT STD_LOGIC;
All_Done_A_Transmissione: OUT STD_LOGIC;
All_Done: OUT STD_LOGIC);

```

```
END COMPONENT;
```

--FSM

```

COMPONENT main_fsm IS
port ( start_main_fsm : in std_logic;
R_PC_IN : IN STD_LOGIC; -- 
RESETn : in std_logic;
CLOCK : STD_LOGIC;
all_done, t_pc_out : out std_logic
);

```

```
end COMPONENT;
```

BEGIN

```

--MEMORY_INTERFACE
DATA_IN_MEM <= OUT_DATA_8_SAMPLER & OUT_GLITCH_8_SAMPLER;

```

```

MAIN_MEMORY : MEMORY_INTERFACE PORT MAP ( DATA_IN_MEM , TRIGGER_EVENT ,
WR_REQ_MEM , RD_REQ_MEM , ALL_DONE_TRASM_MEM , CLOCK , RESETN ,

```

```
DATA_IN_OUT_SRAM, ADDRESS, CS_SRAM, WE_SRAM, OE_SRAM , DATA_OUT_MEM ,  
ALL_DONE_AW, ALL_DONE_AR, WR_DONE, RD_DONE );
```

```
DATA_IN_pC_INTERFACE <= DATA_OUT_MEM;
```

-SAMPLER

```
SAMPLER : SAMPLE8 PORT MAP ( CLOCK , IN_DATA_8_SAMPLER , SAMPLE_8_SAMPLER ,  
RESETN_8_SAMPLER , PS_8_SAMPLER, OUT_DATA_8_SAMPLER, OUT_GLITCH_8_SAMPLER  
) ;
```

-TRIGGER

```
TRIGGER: TRIGGER PORT MAP ( TRIGGER_REFERENCE, SAMPLES , TRIGGER_SAMPLE ,  
START_TRIGGER, CLOCK , RESETN_TRIGGER , START_ACQUISITION, DONE_TRIGGER);
```

-PC_INTERFACE

```
ALL_DONE_S_PC <= ALL_DONE_AW;
```

```
TC256_PC <= ALL_DONE_AR;
```

```
start_trasm_dato_pc <= rd_done;
```

```
PC_INTERFACE : PC_INTERFACE PORT MAP ( STARTER_PC , CLOCK, RESETN_PC ,  
DATA_IN_PC_INTERFACE , TC256_PC , ALL_DONE_S_PC, R_PC_in , start_trasm_dato_pc ,  
glitch_in_pc , s_pc , r_pc , tr , f_pc , out_t , all_done_a_trasmissione , all_done);
```

end behaviour;

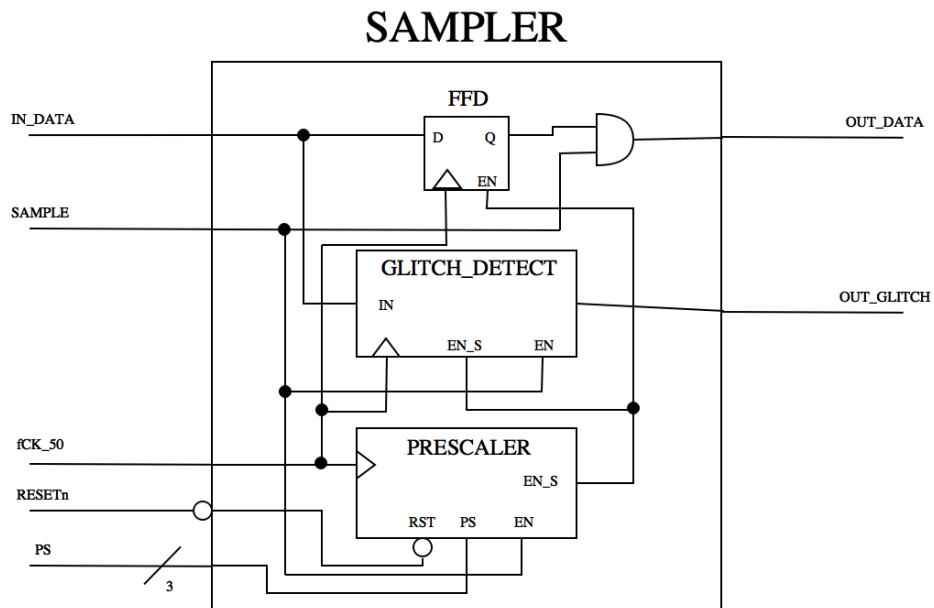
Sampler

Blocco realizzato in logica combinatoria e sequenziale senza control unit.

Lo scopo di questo blocco è campionare l'ingresso e rilevare l'eventuale glitch, comunicandolo alla macchina attraverso una linea dedicata per ogni canale di campionamento (OUT_GLITCH). Tutto questo a varie frequenze di campionamento, regolabili dall'utente.

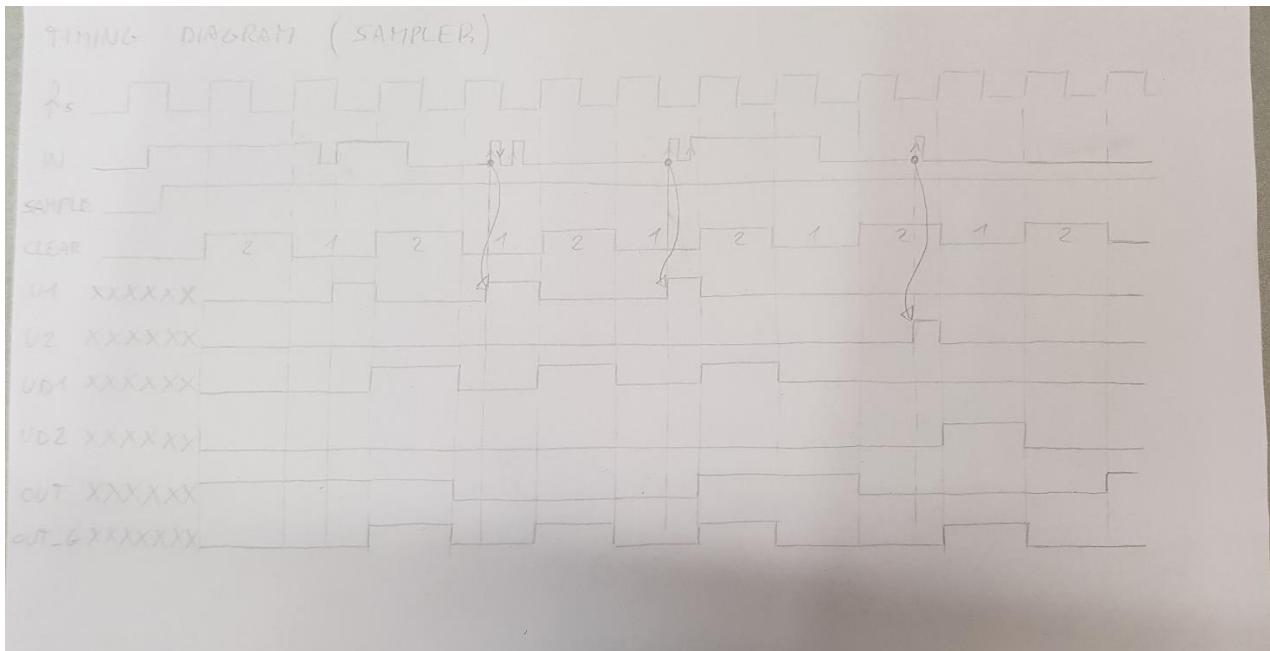
Il circuito più generale è sintetizzato come raffigurato nel datapath. Nelle successive sezioni GLITCH_DETECT e PRESCALER sono mostrati i circuiti nel dettaglio.

Datapath



Datapath fisico del sampler

Timing



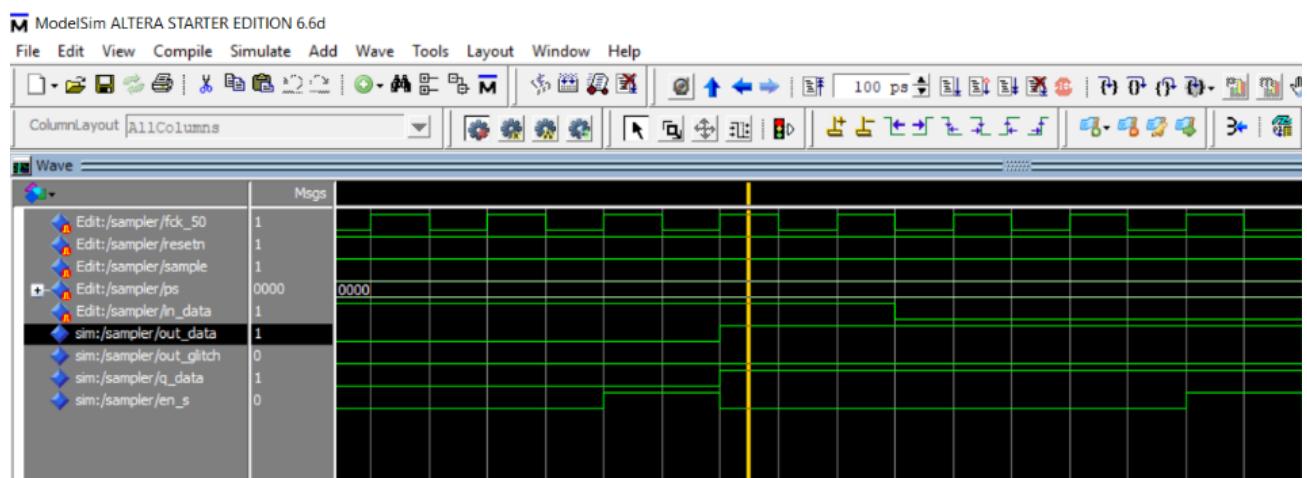
Timing del sampler

Simulazioni

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

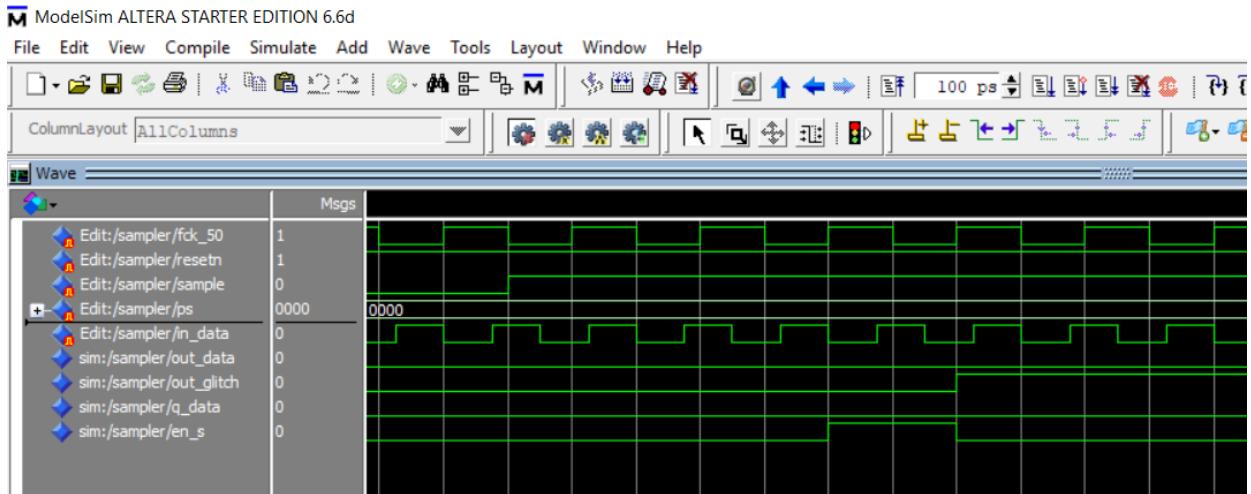
Sono state effettuate 3 simulazioni sul blocco sampler.

Ci siamo messi nel caso di un segnale di ingresso senza glitch.



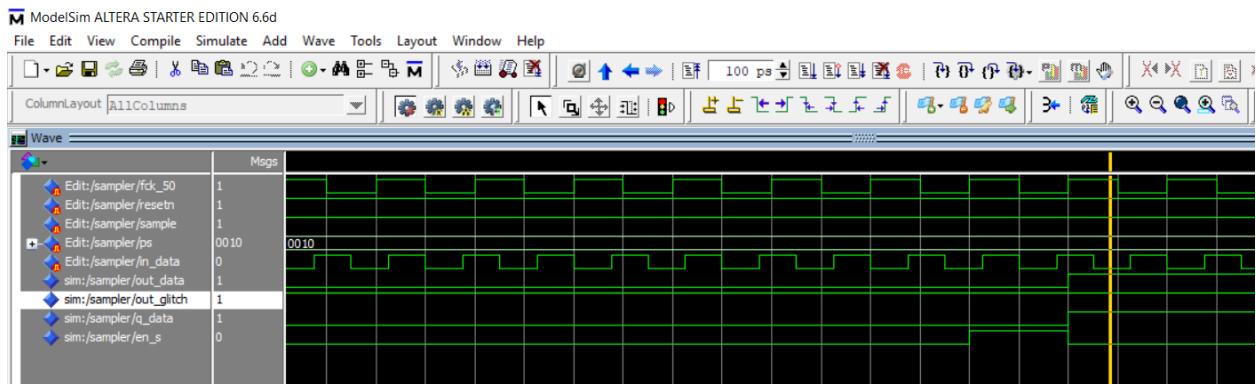
Prima simulazione del sampler

Successivamente abbiamo provato nel caso di un segnale in ingresso con glitch.



Seconda simulazione del sampler

Infine per verificare il funzionamento del prescaler abbiamo provato a fissarlo a '2' (codice binario: 0010).



Terza simulazione del sampler

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

File VHDL: sampler.vhdl

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity sampler is
Port (
    in_data : IN STD_LOGIC;
    sample : IN STD_LOGIC;
    fck_50 : IN STD_LOGIC;
    resetn : IN STD_LOGIC;

```

```

    ps : IN STD_LOGIC_VECTOR(3 DOWNTO 0);

    out_data : OUT STD_LOGIC;
    out_glitch : OUT STD_LOGIC);
end sampler;

```

architecture Behaviour of sampler is

```

COMPONENT flipflopD_up IS
PORT (
    D, Clock, Resetn, Load : IN STD_LOGIC;
    Q :OUT STD_LOGIC);
END COMPONENT;

```

```

COMPONENT glitch_detector is
PORT (      in_signal : IN STD_LOGIC;
            fck_50 : IN STD_LOGIC;
            en_s : IN STD_LOGIC;
            sample : IN STD_LOGIC;

            out_glitch : OUT STD_LOGIC);
END COMPONENT;

```

```

COMPONENT Prescaler is
PORT (      fck_50 : IN STD_LOGIC;
            ps : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            enable : IN STD_LOGIC;
            resetn : IN STD_LOGIC;

            en_s : OUT STD_LOGIC);
END COMPONENT;

```

```

--SIGNAL INTERNAL
SIGNAL Q_data : STD_LOGIC;
SIGNAL en_s : STD_LOGIC;

```

```

-- DATAPATH OF GLITCH_DETECTOR
BEGIN

```

-acquisition data

```

    FFD_data : flipflopD_up PORT MAP (in_data, fck_50, resetn, en_s, Q_data);
    out_data <= Q_data AND sample;

```

-acquisition glitch

```

    glitch_detector1 : glitch_detector PORT MAP (in_data, fck_50, en_s, sample, out_glitch);

```

-prescaler

```

    prescaler1 : Prescaler PORT MAP(fck_50, ps, sample, resetn, en_s);

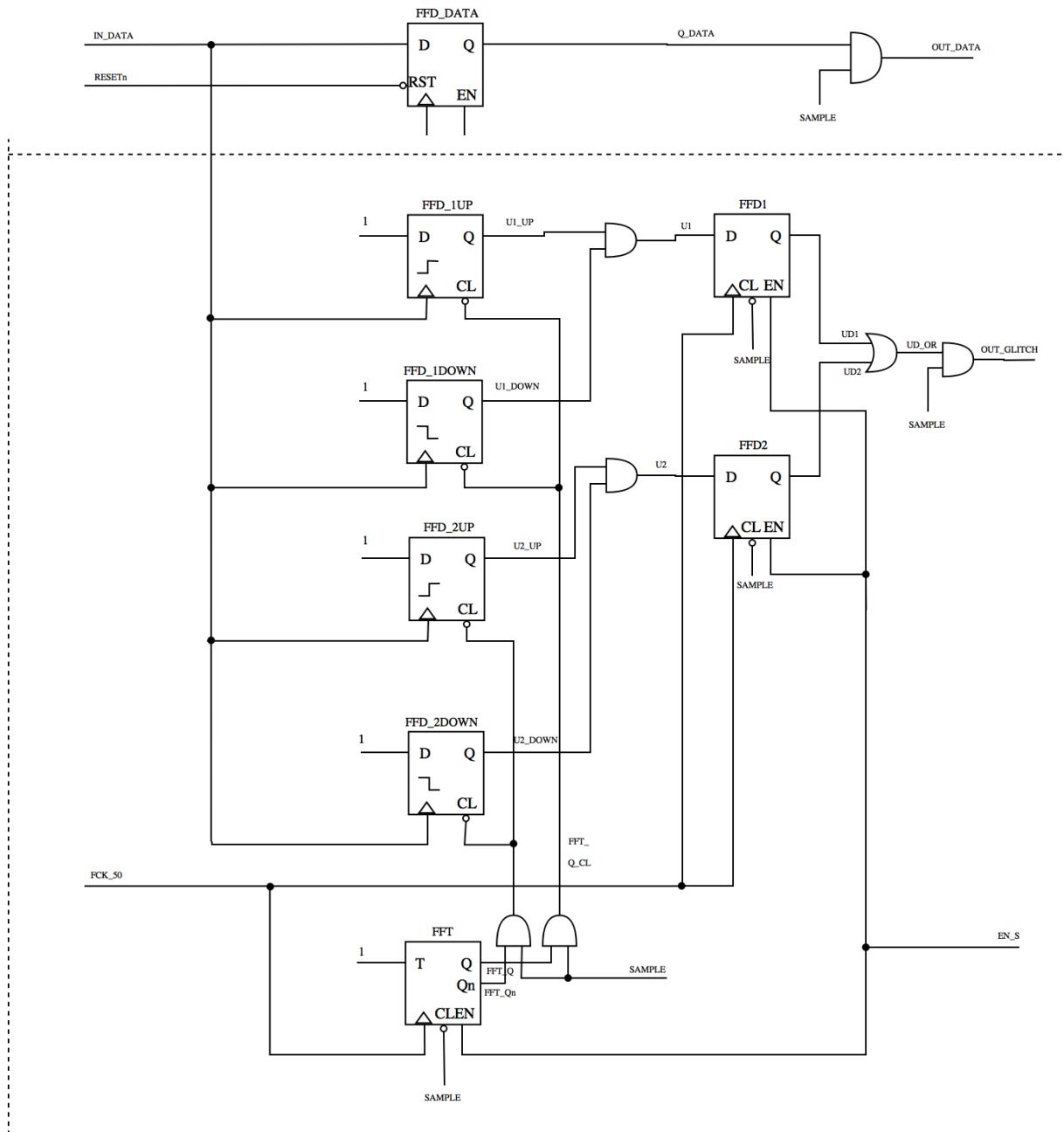
```

END Behaviour;

Glitch detector

Questo blocco, descritto anche nel capitolo sampler, si occupa di rilevare i glitch. E' stato necessario utilizzare un circuito sequenziale asincrono rispetto al clock dato che era necessario che per alcuni flipflop il segnale che arriva dall'ingresso come attivatore non fosse il clock come finora avevamo studiato. Nel datapath viene illustrato il circuito combinatorio e sequenziale utilizzato.

Datapath



Datapath del glitch detector e del campionatore di segnale

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: glitch_detector.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity glitch_detector is
Port (  in_signal : IN STD_LOGIC;
        fck_50 : IN STD_LOGIC;
        en_s : IN STD_LOGIC;
        sample : IN STD_LOGIC;
        out_glitch : OUT STD_LOGIC);
end glitch_detector;
```

architecture Behaviour of glitch_detector is

```
COMPONENT flipflopT IS
PORT(
        T, Clock , Resetn, Load : IN STD_LOGIC;
        Q, Qn : OUT STD_LOGIC);
END COMPONENT;
```

```
COMPONENT flipflopD_up IS
PORT (
        D, Clock, Resetn, Load : IN STD_LOGIC;
        Q :OUT STD_LOGIC);
END COMPONENT;
```

```
COMPONENT flipflopD_down IS
PORT (
        D, Clock, Resetn, Load : IN STD_LOGIC;
        Q :OUT STD_LOGIC);
END COMPONENT;
```

-SIGNAL INTERNAL

```
SIGNAL u1_up, u1_down, u1, ud1 : STD_LOGIC;
SIGNAL u2_up, u2_down, u2, ud2 : STD_LOGIC;
SIGNAL ud_or : STD_LOGIC;
SIGNAL fft_q, fft_qn, fft_q_cl, fft_qn_cl : STD_LOGIC;
```

-- DATAPATH OF GLITCH_DETECTOR

BEGIN

```

FFD_1_up : flipflopD_up PORT MAP ('1', in_signal, fft_q_cl, '1', u1_up);
FFD_1_down : flipflopD_down PORT MAP ('1', in_signal, fft_q_cl, '1', u1_down);
u1 <= u1_up AND u1_down;
FFD_1_en : flipflopD_up PORT MAP (u1, fck_50, sample, en_s, ud1);
fft_q_cl <= fft_q AND sample;

FFD_2_up : flipflopD_up PORT MAP ('1', in_signal, fft_qn_cl, '1', u2_up);
FFD_2_down : flipflopD_down PORT MAP ('1', in_signal, fft_qn_cl, '1', u2_down);
u2 <= u2_up AND u2_down;
FFD_2_en : flipflopD_up PORT MAP (u2, fck_50, sample, en_s, ud2);
fft_qn_cl <= fft_qn AND sample;

ud_or <= ud1 OR ud2;
out_glitch <= ud_or AND sample;

FFT : flipflopT PORT MAP ('1', fck_50, sample, en_s, fft_q, fft_qn);

```

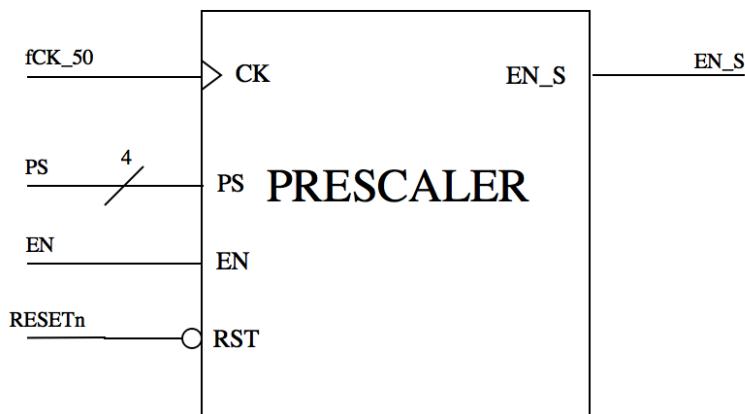
END Behaviour;

Prescaler

Il blocco prescaler si occupa di generare, internamente ad ogni sampler, i segnali di load per i registri che campioneranno i dati in ingresso.

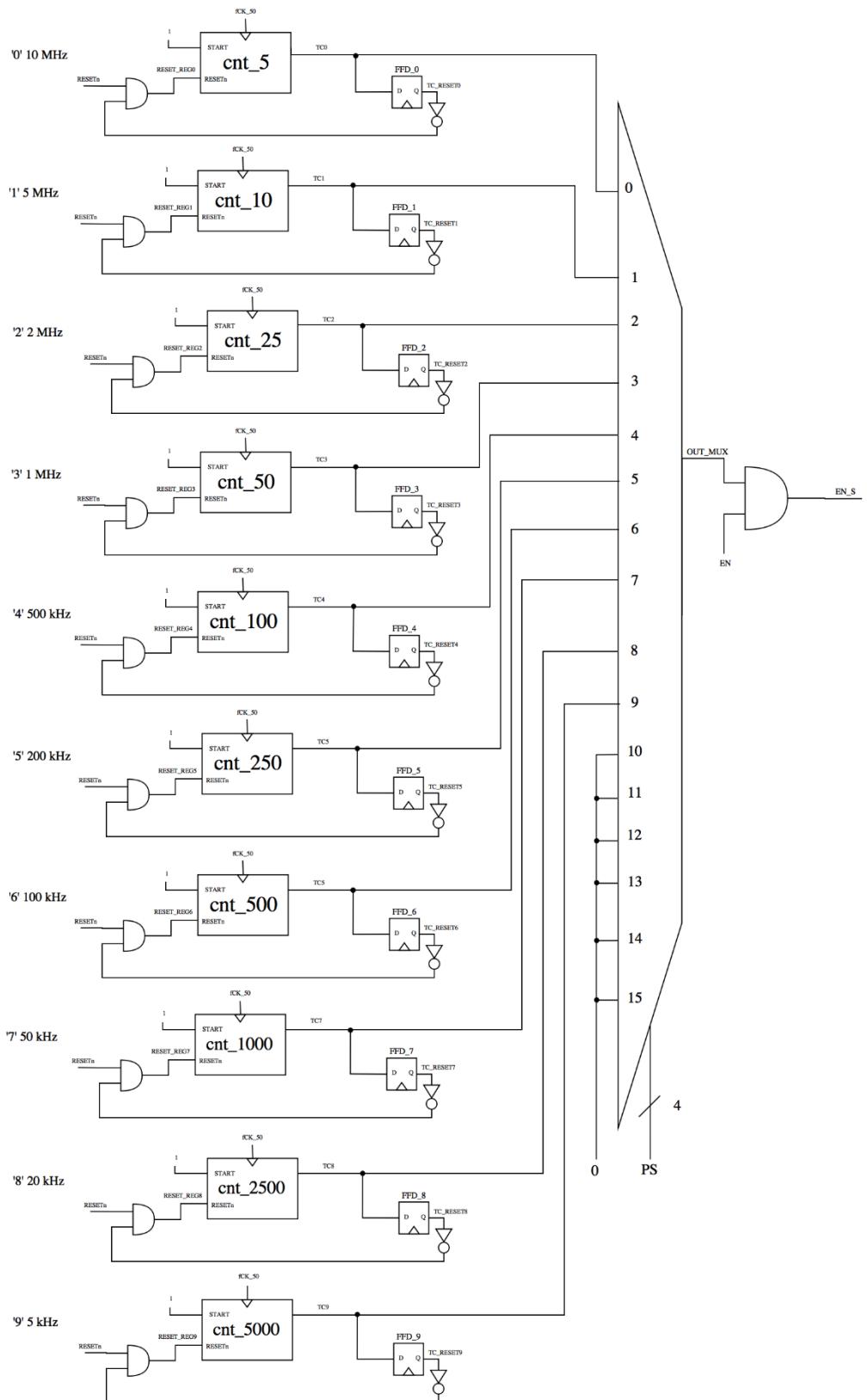
Le specifiche chiedevano un campionamento a varie frequenze partendo da 10 MHz fino ad arrivare a 5 kHz.

Il blocco è sintetizzato come segue.



Blocco prescaler

Datapath

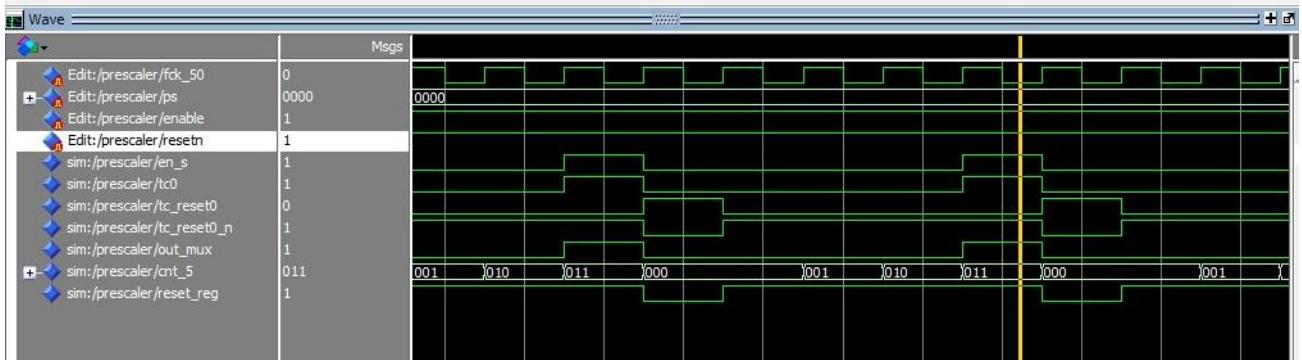


Datapath del prescaler

Simulazioni

Una simulazione del blocco prescaler si può fare impostando PS = 0000 ad ottenere il campionamento del segnale ogni 5 colpi di clock (campionamento a 10 MHz) avendo come clock macchina 50 MHz.

In questo modo si può osservare il lavoro degli elementi a generare l'uscita out_mux secondo quelle che sono le richieste.



Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

File VHDL: prescaler.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity Prescaler is
Port (  fck_50 : IN STD_LOGIC;
          ps : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          enable : IN STD_LOGIC;
          resetn : IN STD_LOGIC;
          en_s : OUT STD_LOGIC);
end Prescaler;

architecture Behaviour of Prescaler is

COMPONENT counterupN IS
GENERIC (N : integer:=13);
PORT(      Enable : IN STD_LOGIC;
          Clock , Resetn : IN STD_LOGIC;
          Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;
```

```

COMPONENT flipflopD_up IS
PORT (
    D, Clock, Resetn, Load : IN STD_LOGIC;
    Q :OUT STD_LOGIC);
END COMPONENT;
```

```

COMPONENT mux16to1 IS
PORT(      a, b, c, d, e, f, g, h, i, l : IN STD_LOGIC;
            s: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            y : OUT STD_LOGIC);
END COMPONENT;
```

-SIGNAL INTERNAL

```

SIGNAL tc0, tc_reset0, tc_reset0_n : STD_LOGIC;
SIGNAL reset_reg0 : STD_LOGIC;
SIGNAL cnt_5 : STD_LOGIC_VECTOR (2 DOWNTO 0);

SIGNAL tc1, tc_reset1, tc_reset1_n : STD_LOGIC;
SIGNAL reset_reg1 : STD_LOGIC;
SIGNAL cnt_10 : STD_LOGIC_VECTOR (3 DOWNTO 0);

SIGNAL tc2, tc_reset2, tc_reset2_n : STD_LOGIC;
SIGNAL reset_reg2 : STD_LOGIC;
SIGNAL cnt_25 : STD_LOGIC_VECTOR (4 DOWNTO 0);

SIGNAL tc3, tc_reset3, tc_reset3_n : STD_LOGIC;
SIGNAL reset_reg3 : STD_LOGIC;
SIGNAL cnt_50 : STD_LOGIC_VECTOR (5 DOWNTO 0);

SIGNAL tc4, tc_reset4, tc_reset4_n : STD_LOGIC;
SIGNAL reset_reg4 : STD_LOGIC;
SIGNAL cnt_100 : STD_LOGIC_VECTOR (6 DOWNTO 0);

SIGNAL tc5, tc_reset5, tc_reset5_n : STD_LOGIC;
SIGNAL reset_reg5 : STD_LOGIC;
SIGNAL cnt_250 : STD_LOGIC_VECTOR (7 DOWNTO 0);

SIGNAL tc6, tc_reset6, tc_reset6_n : STD_LOGIC;
SIGNAL reset_reg6 : STD_LOGIC;
SIGNAL cnt_500 : STD_LOGIC_VECTOR (8 DOWNTO 0);

SIGNAL tc7, tc_reset7, tc_reset7_n : STD_LOGIC;
SIGNAL reset_reg7 : STD_LOGIC;
SIGNAL cnt_1000 : STD_LOGIC_VECTOR (9 DOWNTO 0);

SIGNAL tc8, tc_reset8, tc_reset8_n : STD_LOGIC;
SIGNAL reset_reg8 : STD_LOGIC;
SIGNAL cnt_2500 : STD_LOGIC_VECTOR (11 DOWNTO 0);
```

```

SIGNAL tc9, tc_reset9, tc_reset9_n : STD_LOGIC;
SIGNAL reset_reg9 : STD_LOGIC;
SIGNAL cnt_5000 : STD_LOGIC_VECTOR (12 DOWNTO 0);

SIGNAL out_mux : STD_LOGIC;

```

-- DATAPATH OF PRESCALER

BEGIN

-- '0' => 10 MHz. Count up to 5

```

counter_5:counterupN
    GENERIC MAP (N => 3)
    PORT MAP ('1', fck_50, reset_reg0, cnt_5 );
TC0 <= NOT cnt_5(2) AND cnt_5(1) AND cnt_5(0);
FFD_0 : flipflopD_up PORT MAP (tc0, fck_50, '1', '1', tc_reset0);
tc_reset0_n <= NOT tc_reset0;
reset_reg0 <= resetn AND tc_reset0_n;

```

-- '1' => 5 MHz. Count up to 10

```

counter_10:counterupN
    GENERIC MAP (N => 4)
    PORT MAP ('1', fck_50, reset_reg1, cnt_10 );
TC1 <= cnt_10(3) AND NOT cnt_10(2) AND NOT cnt_10(1) AND NOT cnt_10(0);
FFD_1 : flipflopD_up PORT MAP (tc1, fck_50, '1', '1', tc_reset1);
tc_reset1_n <= NOT tc_reset1;
reset_reg1 <= resetn AND tc_reset1_n;

```

-- '2' => 2 MHz. Count up to 25

```

counter_25:counterupN
    GENERIC MAP (N => 5)
    PORT MAP ('1', fck_50, reset_reg2, cnt_25 );
TC2 <= cnt_25(4) AND NOT cnt_25(3) AND cnt_25(2) AND cnt_25(1) AND cnt_25(0);
FFD_2 : flipflopD_up PORT MAP (tc2, fck_50, '1', '1', tc_reset2);
tc_reset2_n <= NOT tc_reset2;
reset_reg2 <= resetn AND tc_reset2_n;

```

-- '3' => 1 MHz. Count up to 50

```

counter_50:counterupN
    GENERIC MAP (N => 6)
    PORT MAP ('1', fck_50, reset_reg3, cnt_50 );
TC3 <= cnt_50(5) AND cnt_50(4) AND NOT cnt_50(3) AND NOT cnt_50(2) AND NOT cnt_50(1)
AND NOT cnt_50(0);
FFD_3 : flipflopD_up PORT MAP (tc3, fck_50, '1', '1', tc_reset3);
tc_reset3_n <= NOT tc_reset3;
reset_reg3 <= resetn AND tc_reset3_n;

```

```

-- '4' => 500 kHz. Count up to 100
counter_100:counterupN
    GENERIC MAP (N => 7)
    PORT MAP ('1', fck_50, reset_reg4, cnt_100 );
    TC4 <= cnt_100(6) AND cnt_100(5) AND NOT cnt_100(4) AND NOT cnt_100(3) AND NOT
cnt_100(2) AND cnt_100(1)
        AND NOT cnt_100(0);
    FFD_4 : flipflopD_up PORT MAP (tc4, fck_50, '1', '1', tc_reset4);
    tc_reset4_n <= NOT tc_reset4;
    reset_reg4 <= resetn AND tc_reset4_n;

-- '5' => 200 kHz. Count up to 250
counter_250:counterupN
    GENERIC MAP (N => 8)
    PORT MAP ('1', fck_50, reset_reg5, cnt_250 );
    TC5 <= cnt_250(7) AND cnt_250(6) AND cnt_250(5) AND cnt_250(4) AND cnt_250(3) AND NOT
cnt_250(2)
        AND NOT cnt_250(1) AND NOT cnt_250(0);
    FFD_5 : flipflopD_up PORT MAP (tc5, fck_50, '1', '1', tc_reset5);
    tc_reset5_n <= NOT tc_reset5;
    reset_reg5 <= resetn AND tc_reset5_n;

-- '6' => 100 kHz. Count up to 500
counter_500:counterupN
    GENERIC MAP (N => 9)
    PORT MAP ('1', fck_50, reset_reg6, cnt_500 );
    TC6 <= cnt_500(8) AND cnt_500(7) AND cnt_500(6) AND cnt_500(5) AND cnt_500(4) AND NOT
cnt_500(3) AND NOT cnt_500(2)
        AND cnt_500(1) AND NOT cnt_500(0);
    FFD_6 : flipflopD_up PORT MAP (tc6, fck_50, '1', '1', tc_reset6);
    tc_reset6_n <= NOT tc_reset6;
    reset_reg6 <= resetn AND tc_reset6_n;

-- '7' => 50 kHz. Count up to 1000
counter_1000:counterupN
    GENERIC MAP (N => 10)
    PORT MAP ('1', fck_50, reset_reg7, cnt_1000 );
    TC7 <= cnt_1000(9) AND cnt_1000(8) AND cnt_1000(7) AND cnt_1000(6) AND cnt_1000(5) AND
NOT cnt_1000(4)
        AND NOT cnt_1000(3) AND cnt_1000(2) AND cnt_1000(1) AND NOT
cnt_1000(0);
    FFD_7 : flipflopD_up PORT MAP (tc7, fck_50, '1', '1', tc_reset7);
    tc_reset7_n <= NOT tc_reset7;
    reset_reg7 <= resetn AND tc_reset7_n;

-- '8' => 20 kHz. Count up to 2500
counter_2500:counterupN
    GENERIC MAP (N => 12)
    PORT MAP ('1', fck_50, reset_reg8, cnt_2500 );

```

```

TC8 <= cnt_2500(11) AND NOT cnt_2500(10) AND NOT cnt_2500(9) AND cnt_2500(8) AND
cnt_2500(7) AND cnt_2500(6)
    AND NOT cnt_2500(5) AND NOT cnt_2500(4) AND NOT cnt_2500(3) AND
NOT cnt_2500(2) AND cnt_2500(1)
    AND NOT cnt_2500(0);
FFD_8 : flipflopD_up PORT MAP (tc8, fck_50, '1', '1', tc_reset8);
tc_reset8_n <= NOT tc_reset8;
reset_reg8 <= resetn AND tc_reset8_n;

-- '9' => 10 kHz. Count up to 5000
counter_5000:counterupN
    GENERIC MAP (N => 13)
    PORT MAP ('1', fck_50, reset_reg9, cnt_5000 );
TC9 <= cnt_5000(12) AND NOT cnt_5000(11) AND NOT cnt_5000(10) AND cnt_5000(9) AND
cnt_5000(8) AND cnt_5000(7)
    AND NOT cnt_5000(6) AND NOT cnt_5000(5) AND NOT cnt_5000(4) AND
NOT cnt_5000(3) AND cnt_5000(2)
    AND NOT cnt_5000(1) AND NOT cnt_5000(0);
FFD_9 : flipflopD_up PORT MAP (tc9, fck_50, '1', '1', tc_reset9);
tc_reset9_n <= NOT tc_reset9;
reset_reg9 <= resetn AND tc_reset9_n;

mux : mux16to1 PORT MAP (TC0,TC1,TC2,TC3,TC4,TC5,TC6,TC7,TC8,TC9, ps, out_mux);

en_s <= out_mux AND enable;

```

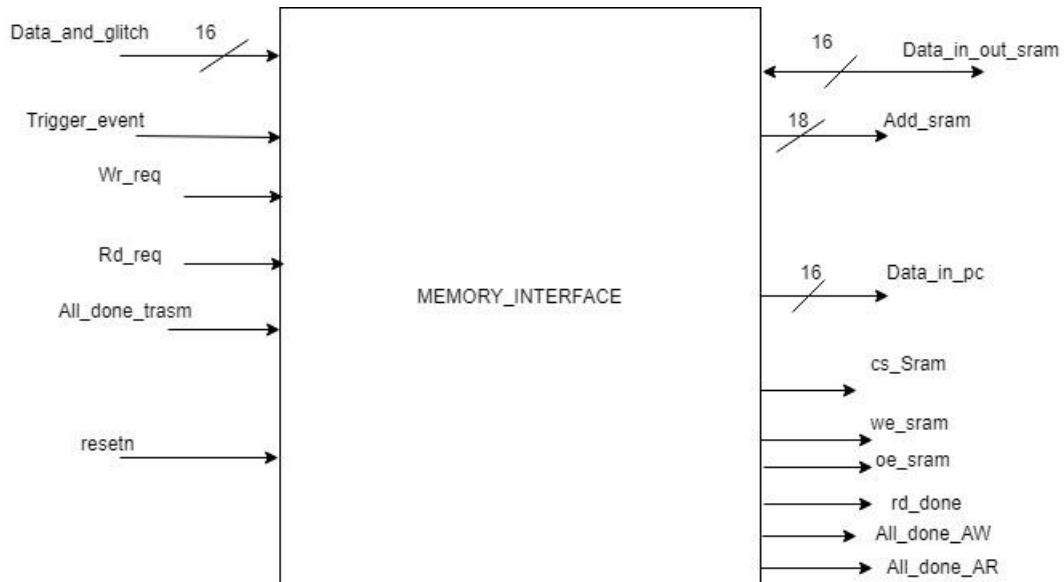
END Behaviour;

Memory interface

Il blocco memory interface si occupa della gestione del protocollo di comunicazione utilizzato con la SRAM. Si occupa quindi della generazione dei segnali di controllo, del flusso dei dati e della gestione degli indirizzi di memoria.

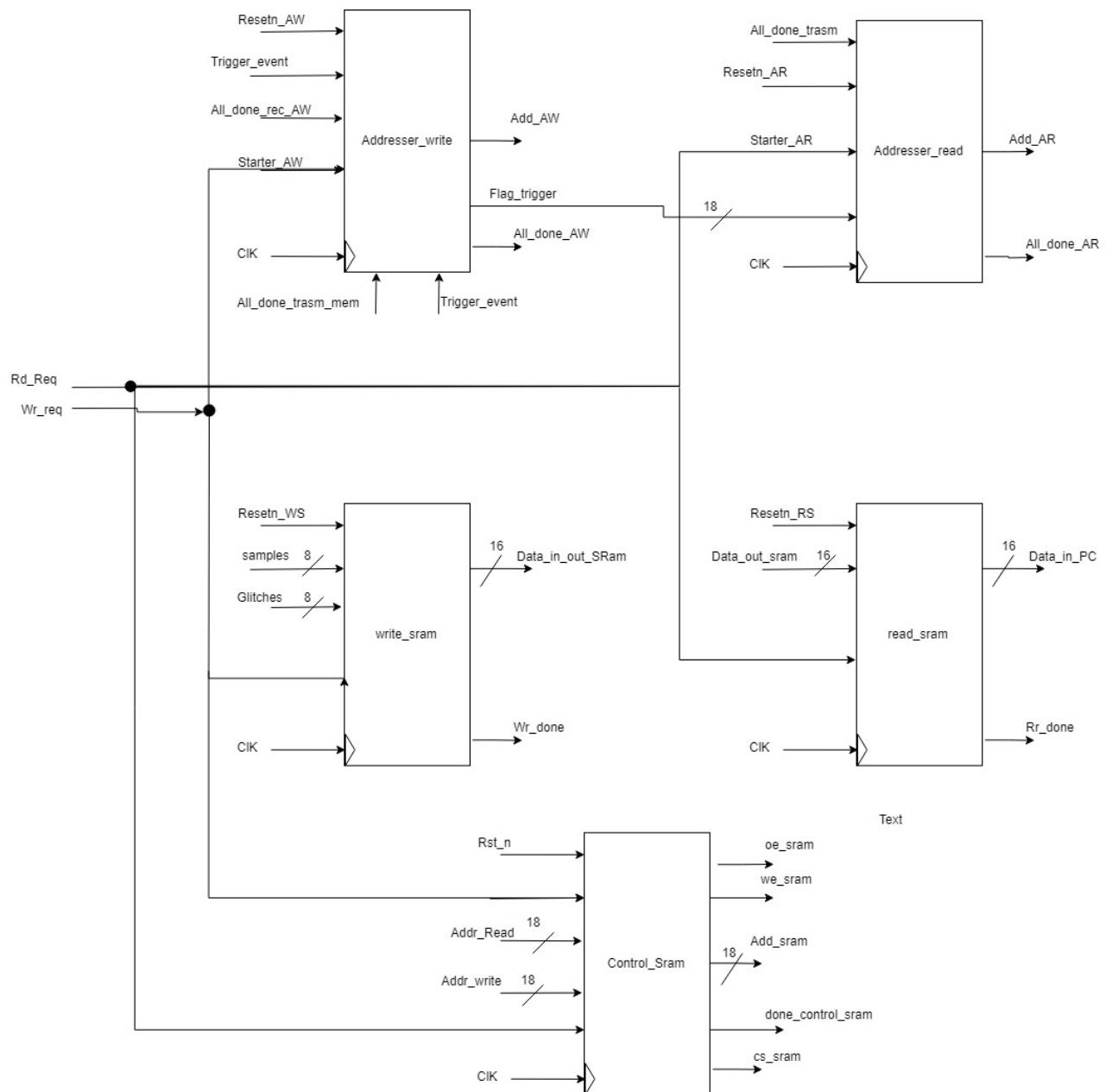
Per l'esecuzione continua è stata progettata una logica con la gestione degli indirizzi come buffer circolare(se ne occupano i blocchi Addresser_write e Addresser_read).

La control_Sram si occupa invece della generazione dei segnali di controllo, write_sram e read_sram regolano il passaggio dei dati.



Blocco della memory interface

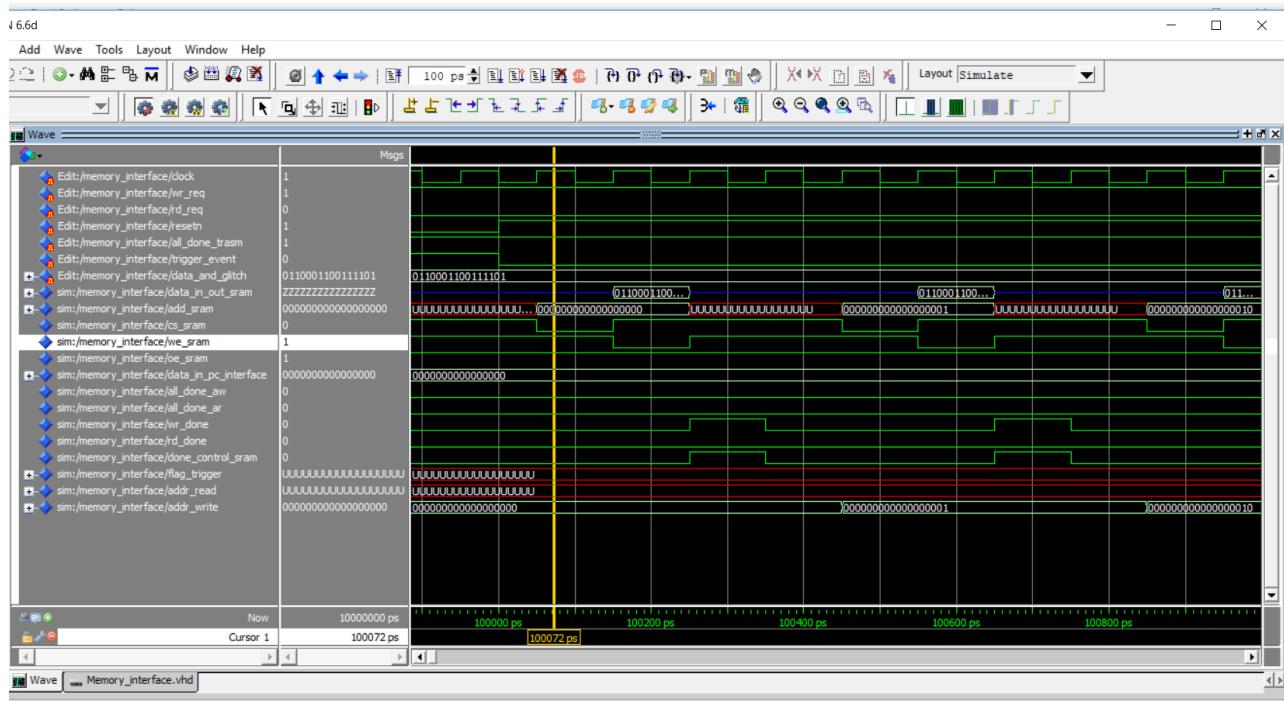
Datapath



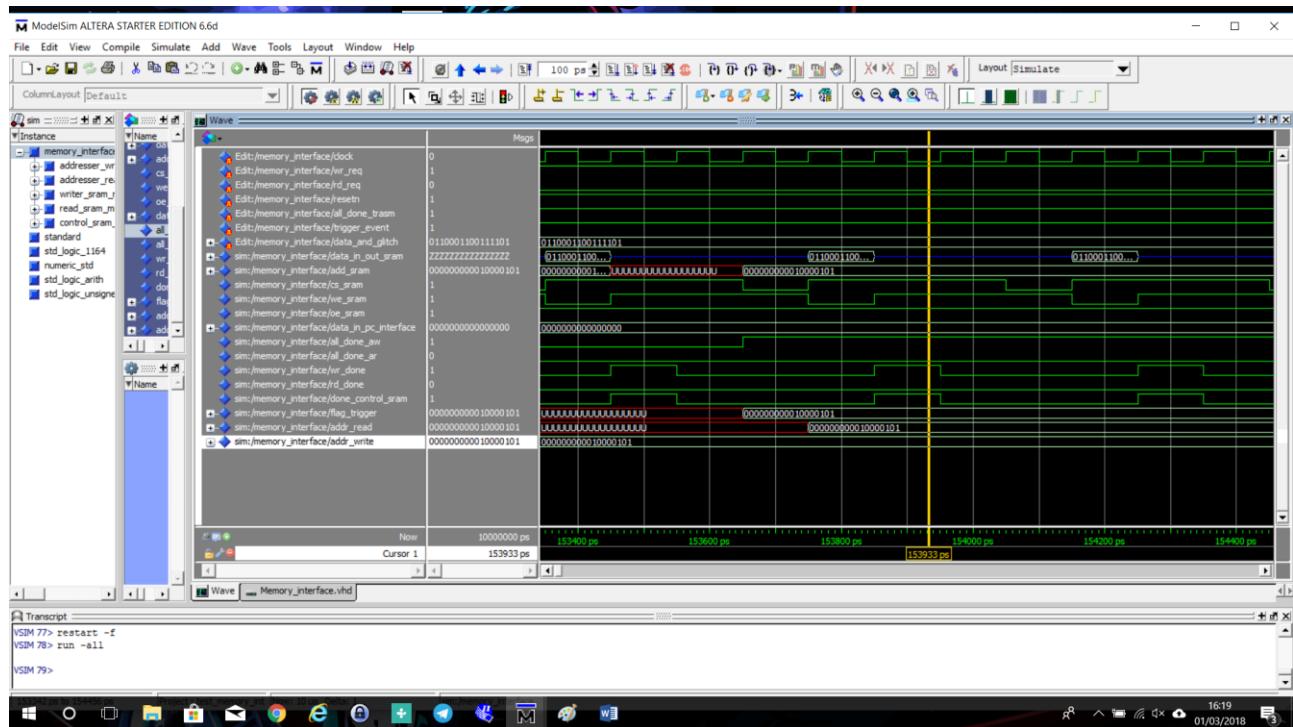
Datapath della memory interface

Simulazioni

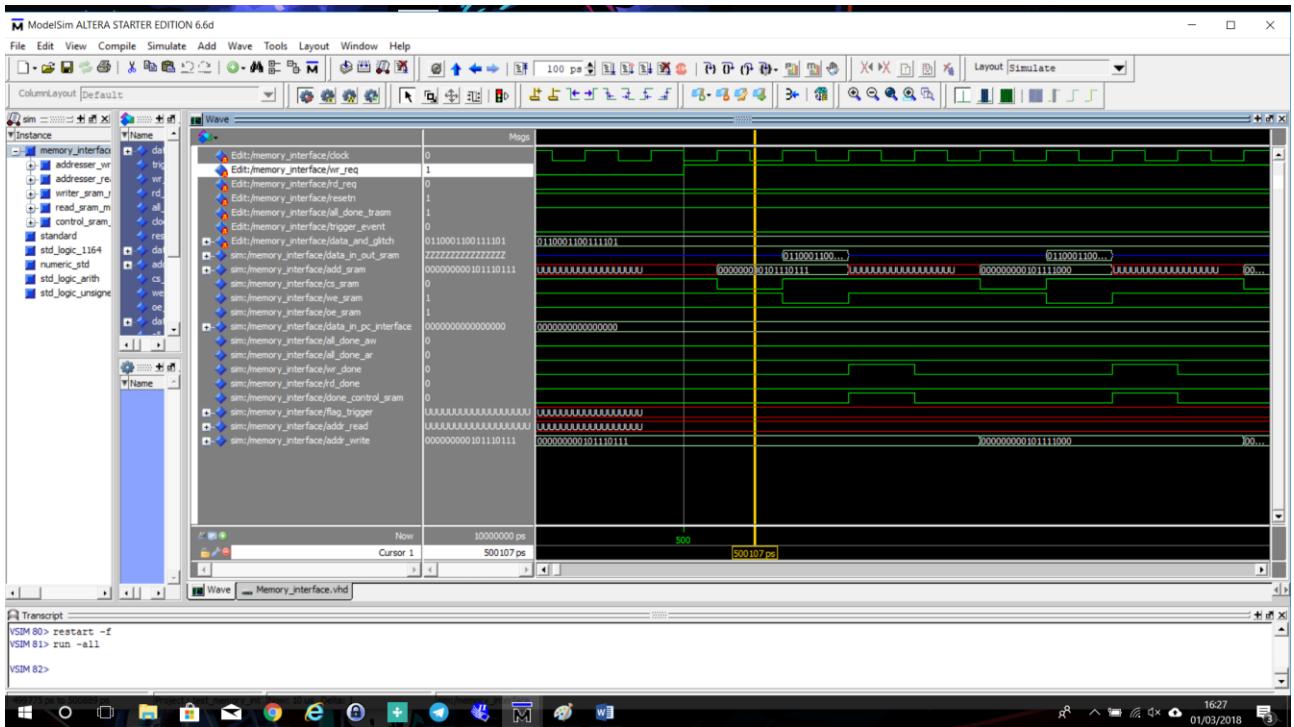
Sono state fatte simulazioni nel ciclo di scrittura e nel ciclo di lettura.



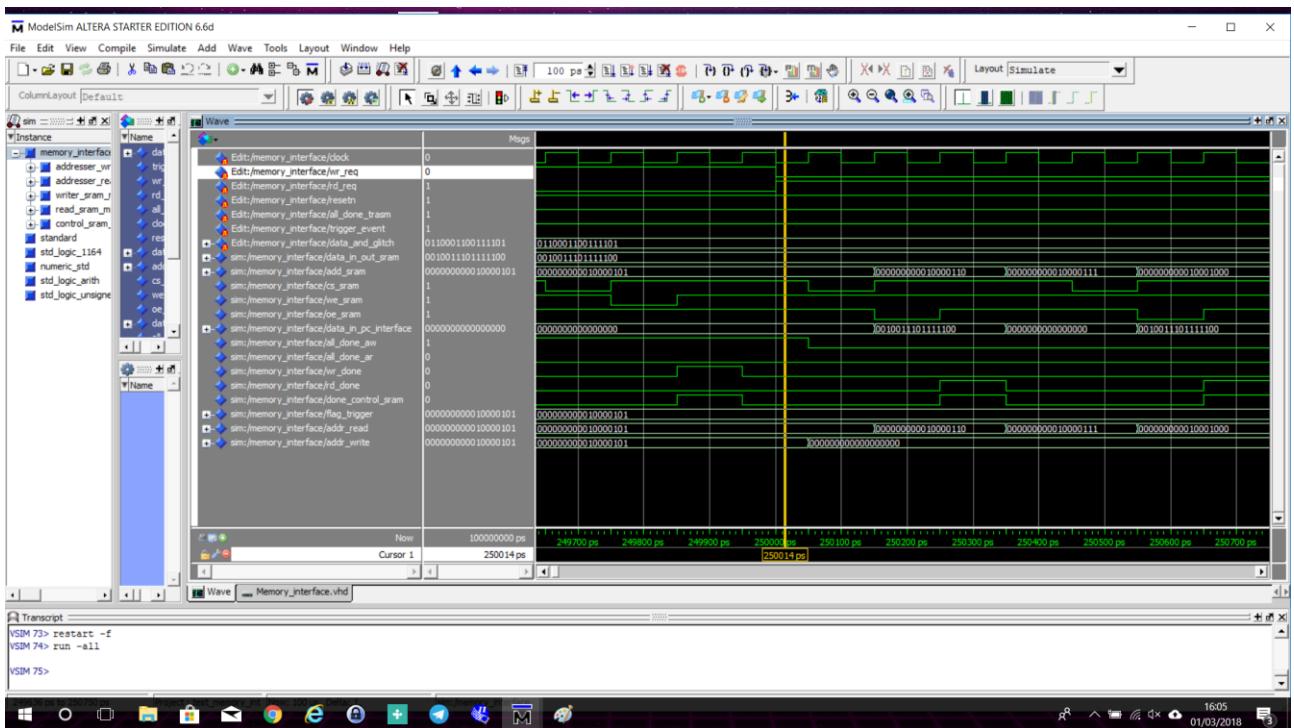
Simulazione della memory interface nel ciclo di scrittura



Simulazione della memory interface nel ciclo di scrittura con segnale di trigger



Simulazione della memory interface nel ciclo di scrittura senza segnale di trigger



Simulazione della memory interface nel ciclo di lettura

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

File VHDL: memory_interface.vhdl

--Memory interface

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY Memory_interface IS
PORT(
    --input
    Data_and_glitch: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Trigger_event: IN STD_LOGIC;
    WR_REQ: IN STD_LOGIC;
    RD_REQ: IN STD_LOGIC;
    All_done_trasm: IN STD_LOGIC;
    Clock, Resetn: IN STD_LOGIC;

    --input and output
    DATA_IN_OUT_SRAM: BUFFER STD_LOGIC_VECTOR(15
DOWNTO 0);

    --output
    Add_SRAM: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
    CS_SRAM, WE_SRAM, OE_SRAM: OUT STD_LOGIC;
    Data_IN_PC_Interface: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    ALL_DONE_AW, ALL_DONE_AR: OUT STD_LOGIC;
    WR_DONE, RD_DONE: OUT STD_LOGIC
);

END Memory_interface;
```

ARCHITECTURE behavior OF Memory_interface IS

```
--SIGNAL
SIGNAL DONE_CONTROL_SRAM: STD_LOGIC;
SIGNAL Flag_trigger, ADDR_READ, ADDR_WRITE: STD_LOGIC_VECTOR(17 DOWNTO 0);
```

--COMPONENTS

```
COMPONENT Writer_SRAM IS
PORT (
wr_request : IN STD_LOGIC;
clock , resetN : IN STD_LOGIC;
SAMPLES , GLITCHES : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
DATA_OUT_BUFFER : OUT STD_LOGIC_VECTOR ( 15 DOWNTO 0);
DONE_WRITE : OUT STD_LOGIC);
END COMPONENT;
```

```

COMPONENT Control_SRAM IS
PORT (
wr_request , rd_request , rstn_sram , clock : in std_logic;
address_read , address_write : in std_logic_vector( 17 downto 0);
address_memory : out std_logic_vector ( 17 downto 0 );
done_control_sram : out std_logic;
cs_sram , we_sram , oe_sram : out std_logic);
END COMPONENT;
```

```

COMPONENT Addresser_read IS
PORT(
Starter,Start_count: IN STD_LOGIC;
Clock,Resetn:IN STD_LOGIC;
Flag_trigger: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
Address: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
All_Done: OUT STD_LOGIC);
END COMPONENT;
```

```

COMPONENT Read_SRAM IS
PORT(
RD_REQ: IN STD_LOGIC;
Clock, Resetn: IN STD_LOGIC;
Data_SRAM: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
Dout: OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
RD_Done: OUT STD_LOGIC);
END COMPONENT;
```

```

COMPONENT Addresser_write IS
PORT(
Starter: IN STD_LOGIC;
Clock, Resetn:IN STD_LOGIC;
All_done_record: IN STD_LOGIC;
Trigger_event: IN STD_LOGIC;
All_Done: OUT STD_LOGIC;
Address : BUFFER STD_LOGIC_VECTOR(17 DOWNTO 0);
Flag_trigger: OUT STD_LOGIC_VECTOR(17 DOWNTO 0));
END COMPONENT;
```

```

BEGIN
--DATAPATH
```

```

Addresser_write_MI : Addresser_write port map (WR_REQ, Clock, Resetn, DONE_CONTROL_SRAM,
Trigger_event, ALL_DONE_AW, ADDR_WRITE, Flag_trigger);
Addresser_read_MI : Addresser_read port map (RD_REQ, All_done_trasm, Clock, Resetn, Flag_trigger,
ADDR_READ, ALL_DONE_AR);
Writer_SRAM_MI : Writer_SRAM port map (WR_REQ,Clock,Resetn,Data_and_glitch(15 DOWNTO
8),Data_and_glitch(7 DOWNTO 0),DATA_IN_OUT_SRAM, WR_DONE);
Read_SRAM_MI : Read_SRAM port map (RD_REQ, Clock, Resetn, DATA_IN_OUT_SRAM,
DATA_IN_PC_interface, RD_DONE);
```

Control_SRAM_MI : Control_SRAM port map (WR_REQ, RD_REQ, Resetn, Clock, ADDR_READ, ADDR_WRITE, ADD_SRAM, DONE_CONTROL_SRAM, CS_SRAM, WE_SRAM, OE_SRAM);

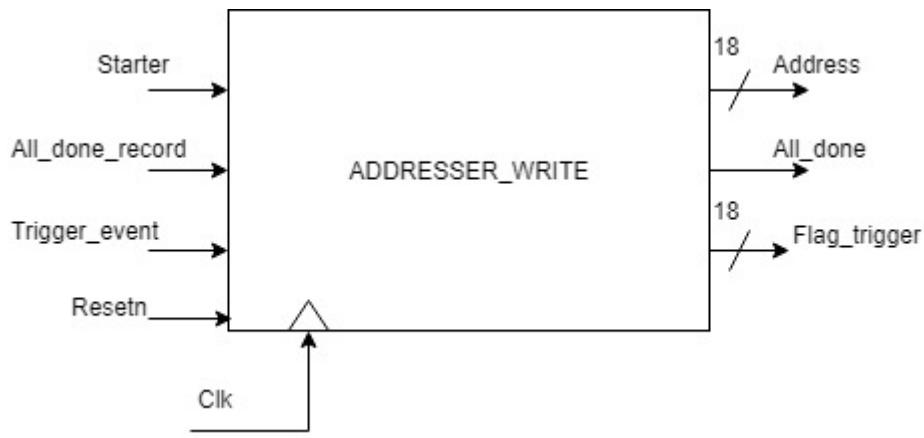
END behavior;

Addresser write

Il blocco Addresser_write ha il compito di generare l'address corretto per la memorizzazione del dato all'interno della sRam.

Passato: Dopo un segnale iniziale di start in uscita viene fornito l'indirizzo “00000000000000000000”. Attende poi che la memorizzazione sia avvenuta all'interno della sRam, segnalata dal controllo All_done_record. In questo caso Aggiorna il valore dell'Address. Finche Trigger_event=0 l'azione descritta si ripete in loop.

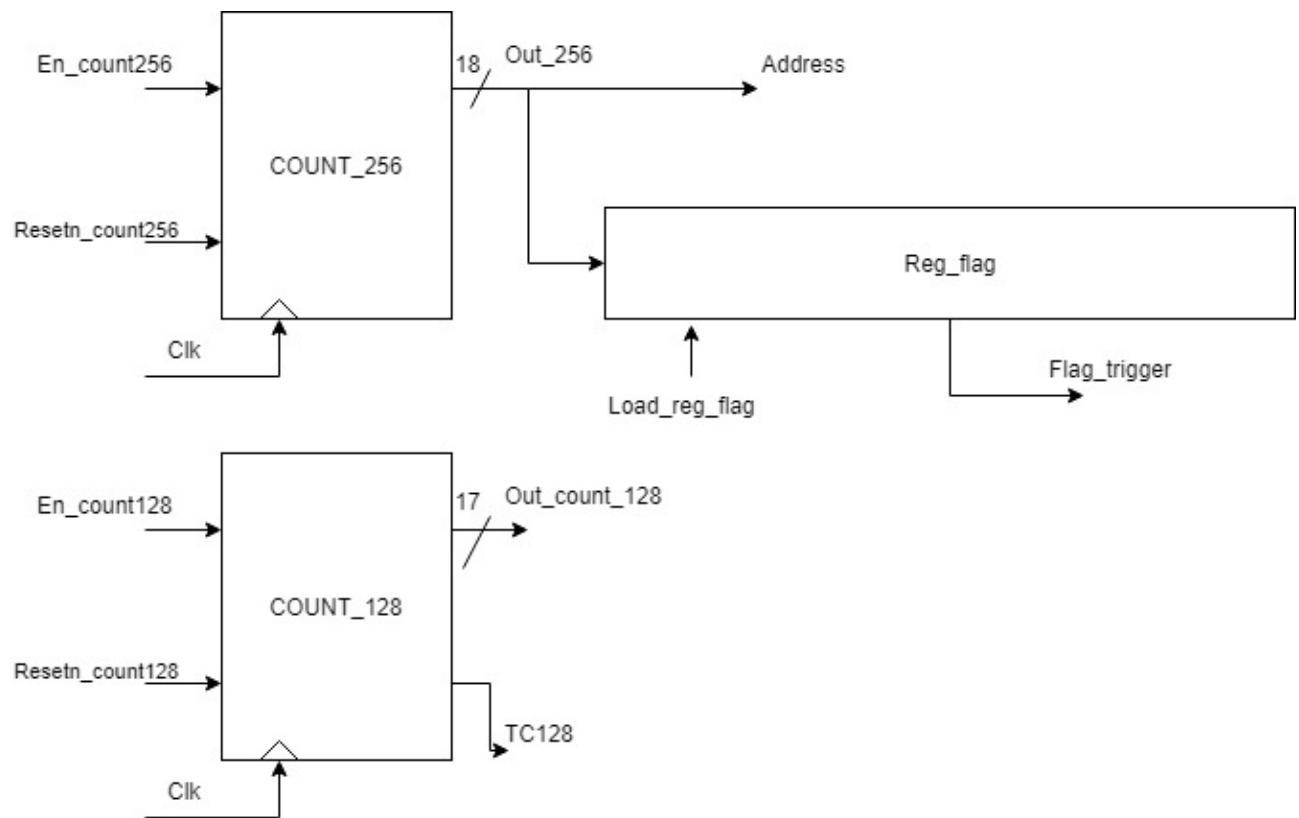
Futuro: In caso si riceva Trigger_event=1 si attiva un secondo contatore che conta fino a metà della grandezza della sRam(128000 indirizzi), continuando a fornire address come prima. Una volta che il secondo contatore giunge al terminal_count viene fornito sul segnale Flag_Trigger il valore dell'ultimo indirizzo (che servirà all'addres_Read per fornire correttamente l'indirizzo per la lettura) e un segnale di all_done.



Pseudocode

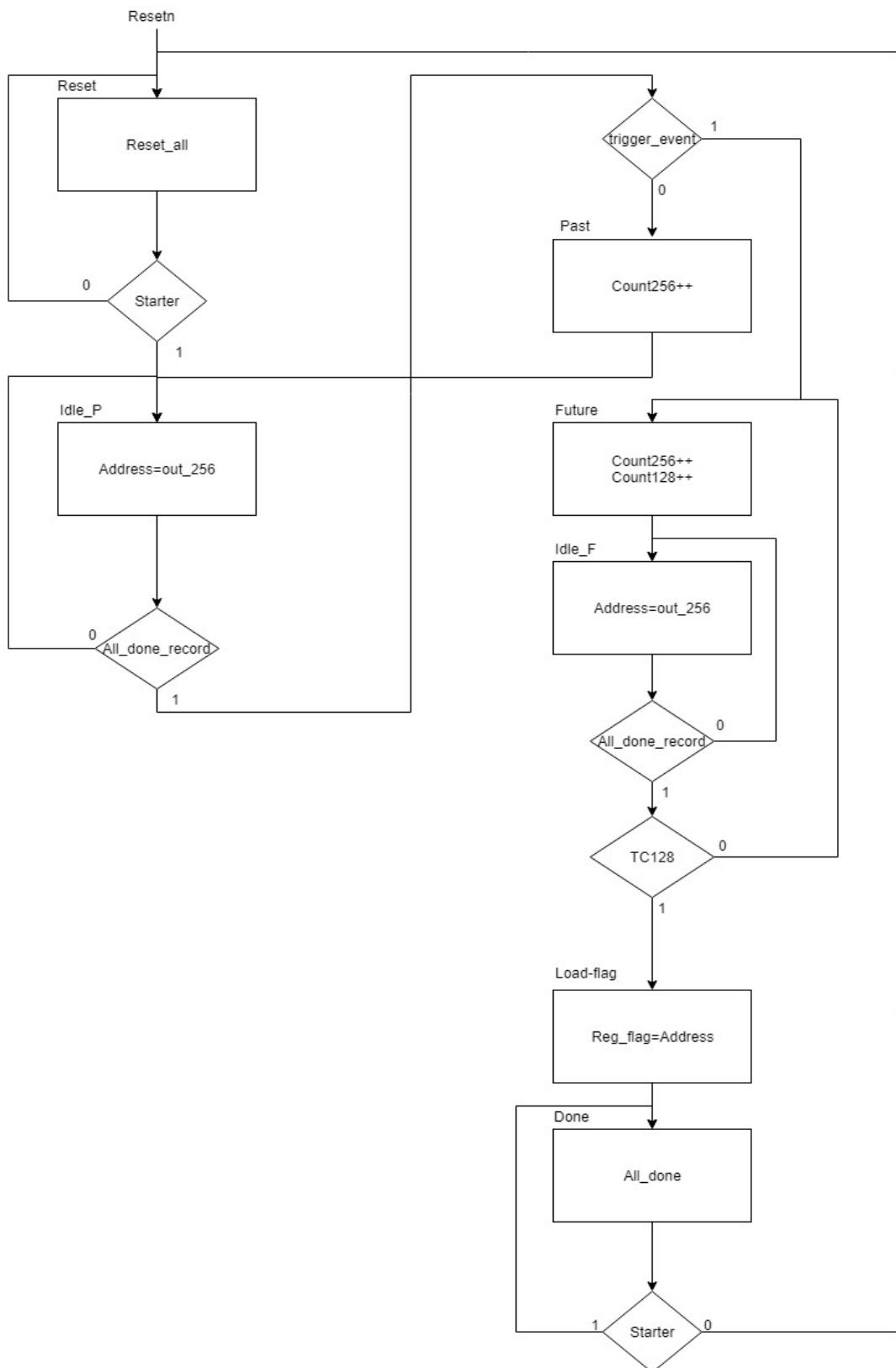
```
Address=0;  
  
If Start=1  
  
  If All_done_record=1  
  
    If Trigger_event=0  
  
      Address++;  
  
    Else  
  
      Flag=Address;  
  
      For(i=Address;i<Address+128000;i++)  
  
        If All_done_record=1  
  
          Address++;  
  
        EndIf;  
  
      EndFor;  
  
      All_done=1;  
  
    EndIf;  
  
  EndIf;  
  
EndIf;
```

Datapath



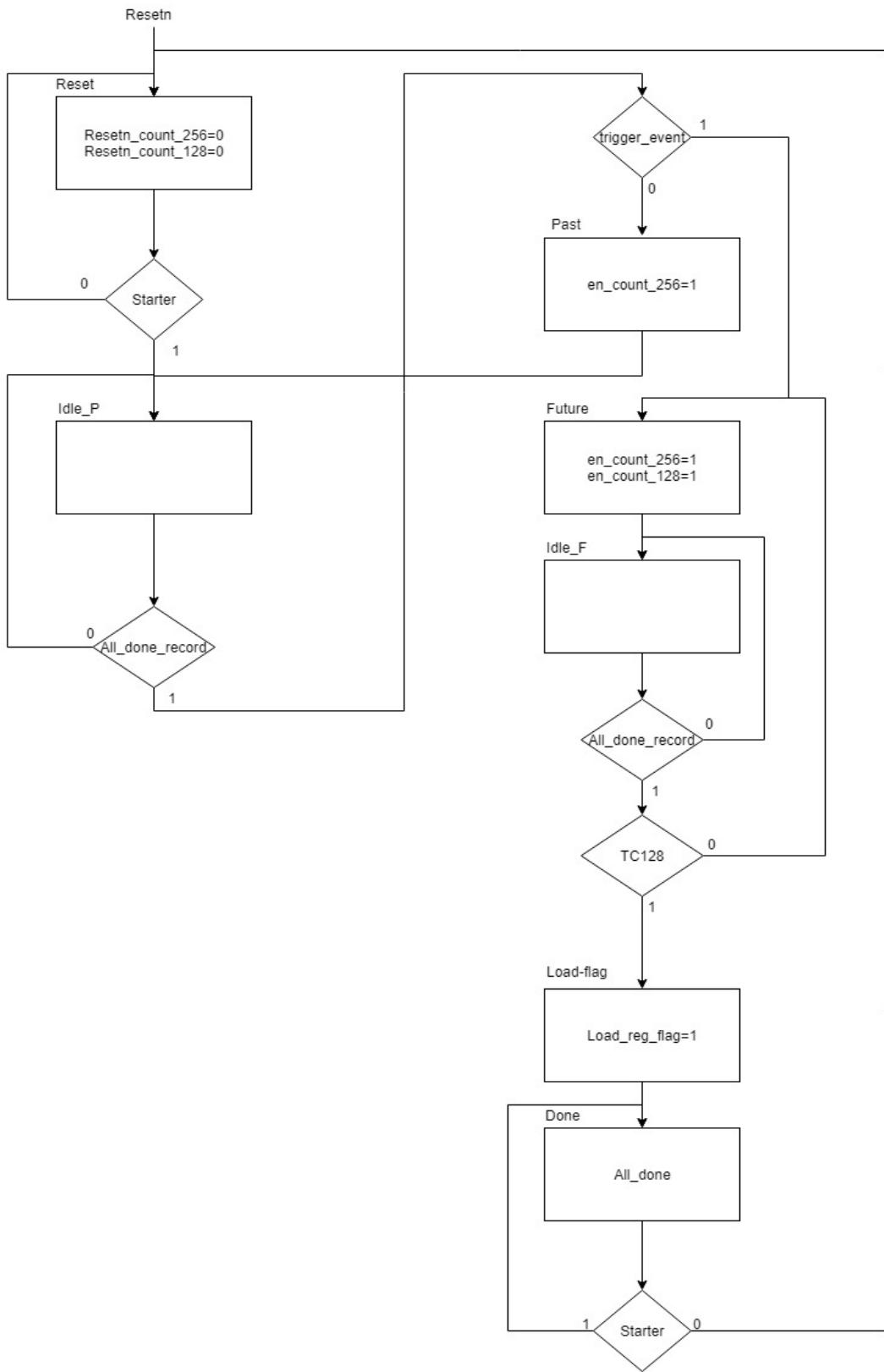
Datapath dell' addresser write

ASM chart



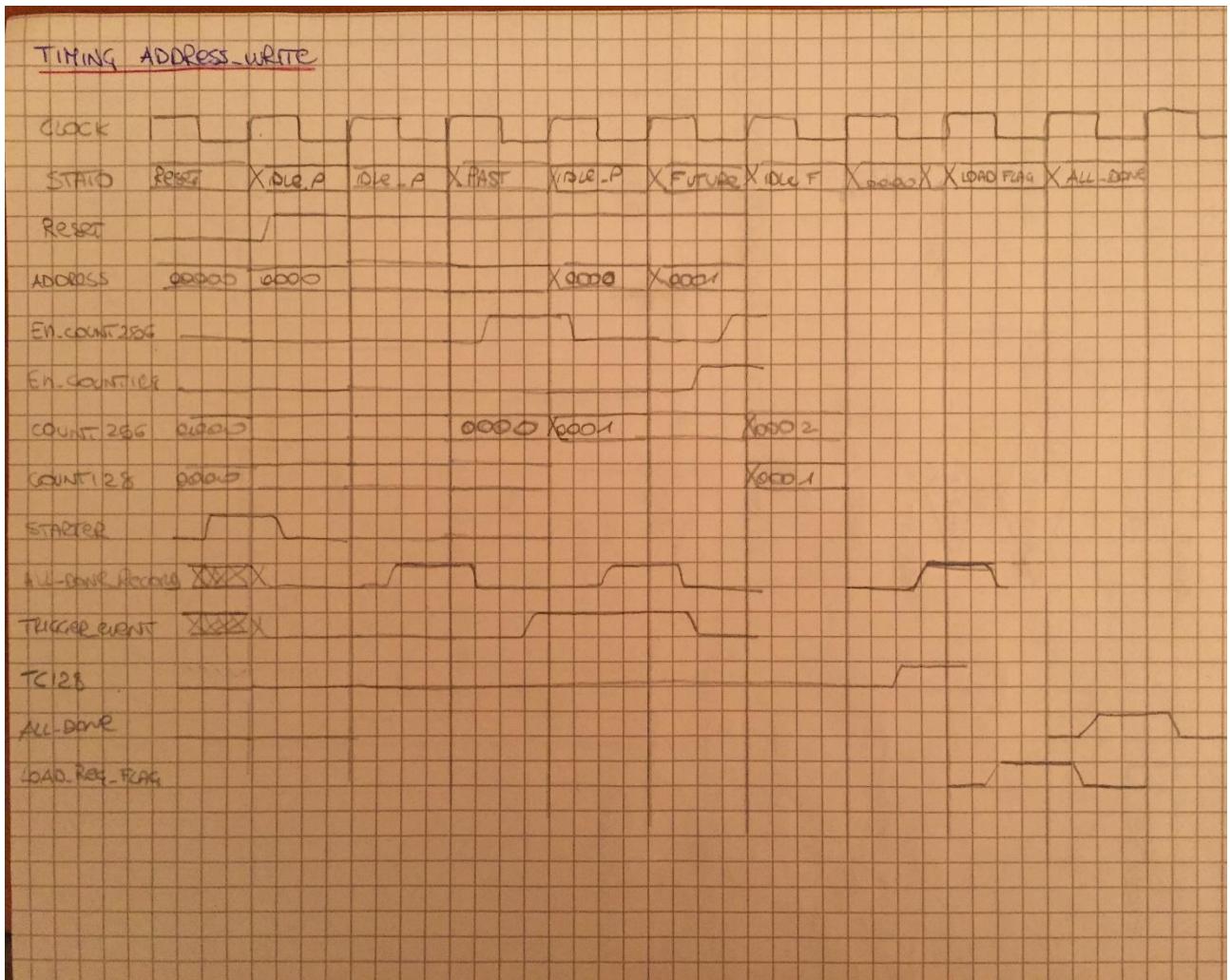
ASM chart dell' addresser write

Control ASM chart



ASM chart dei controlli dell' addresser write

Timing



Timing dell' addresser write

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: addresser_write.vhdl

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;
```

```
ENTITY Addresser_write IS
--PORT( Starter: IN STD_LOGIC;
--      Clock,Resetn:IN STD_LOGIC;
--      All_done_record: IN STD_LOGIC;
--      Trigger_event: IN STD_LOGIC;
--      All_Done: OUT STD_LOGIC);
```

```

-- Address,Flag_trigger: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
--END trasmettitore;
PORT( SW: IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      CLOCK_50: IN STD_LOGIC;
      LEDG: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      LEDR: OUT STD_LOGIC_VECTOR(17 DOWNTO 0));
END Addresser_write;

```

ARCHITECTURE behavior OF Addresser_write IS

-SIGNAL

```

SIGNAL starter,clock,resetn,all_done_record,trigger_event,All_done: std_logic;
SIGNAL address,flag_trigger: std_logic_vector(17 downto 0);
TYPE State_type IS (Reset,Idle_p,Past, Future,Idle_F,load_flag,Done);
SIGNAL y : State_type;
SIGNAL resetn_count256,en_count256,resetn_count128,en_count128,tc128,load_reg_flag: STD_LOGIC;
SIGNAL out_count128 : std_logic_vector(16 downto 0);
SIGNAL out_256:std_logic_vector(17 downto 0);

```

-COMPONENT

```

COMPONENT counterupN IS
  GENERIC (N : integer:=4);
  PORT(   Enable : IN STD_LOGIC;
          Clock , Resetn : IN STD_LOGIC;
          Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;

```

COMPONENT reg_asynch_reset_18bit is

```

port
(
    clk           : in std_logic;
    resetn       : in std_logic;
    pr_in        : in std_logic_vector(17 DOWNTO 0);
    load_parallel : in std_logic;
    pr_out       : out std_logic_vector(17 downto 0)
);

```

end COMPONENT;

BEGIN

FSM_transitions:

```

PROCESS ( Clock , Resetn )
--VARIABLE first_state : std_logic :='0';

```

BEGIN

IF Resetn = '0' THEN

 y <= Reset;

ELSIF (Clock 'EVENT AND Clock = '1') THEN

 CASE y IS

 WHEN Reset => IF starter = '1' THEN y <= Idle_p ;

```

        ELSE y <= Reset ;
        END IF ;
WHEN Idle_p => IF All_done_record = '1' THEN IF trigger_event='1' THEN y <= Future ;
                ELSE y <= Past ;
                END IF ;
        ELSE y<=Idle_p;
        END IF;
WHEN Past => y <= Idle_p ;
WHEN Future => y<=Idle_f;
WHEN Idle_f => IF All_done_record = '1' THEN IF TC128='1' THEN y <= Load_flag ;
                ELSE y <= Future ;
                END IF ;
        ELSE y<=Idle_F;
        END IF;
WHEN Load_flag => y <= Done ;
WHEN Done => IF Starter = '1' THEN y <= Done ;
                ELSE y <= Reset ;
                END IF ;
        END CASE;
    END IF;
END PROCESS;

```

```

FSM_outputs: PROCESS(y)
BEGIN
resetn_count256<='1';
resetn_count128<='1';
Load_reg_flag<='0';
en_count256<='0';
en_count128<='0';
All_Done<='0';

CASE y IS
WHEN Reset=> Resetn_count128<='0';
        Resetn_count256<='0';
WHEN Idle_p => Resetn_count128<='1';
WHEN Past => en_count256<='1';
WHEN Future  =>en_count256<='1';
        en_count128<='1';
WHEN Idle_f => Resetn_count128<='1';
        WHEN Load_flag => Load_reg_flag<='1';
WHEN Done => All_Done <= '1';
        END CASE;
END PROCESS;

```

```

Count256:counterupN
  GENERIC MAP (N => 18)
  PORT MAP (en_count256 , Clock , Resetn_count256 ,Address);

```

Count128:counterupN

```

  GENERIC MAP (N => 17)
  PORT MAP (en_count128 , Clock , Resetn_count128 ,out_count128);
  TC128<= out_count128(16) AND out_count128(15) AND out_count128(14) AND out_count128(13)
AND out_count128(12) AND NOT out_count128(11) AND out_count128(10) AND NOT
out_count128(9) AND NOT out_count128(8) AND NOT out_count128(7) AND NOT out_count128(6)
AND NOT out_count128(5) AND NOT out_count128(4) AND NOT out_count128(3) AND NOT
out_count128(2) AND NOT out_count128(1) AND NOT out_count128(0);

```

```

Reg_flag: reg_asynch_reset_18bit
  PORT MAP(Clock,'1',Address,load_reg_flag,flag_trigger);

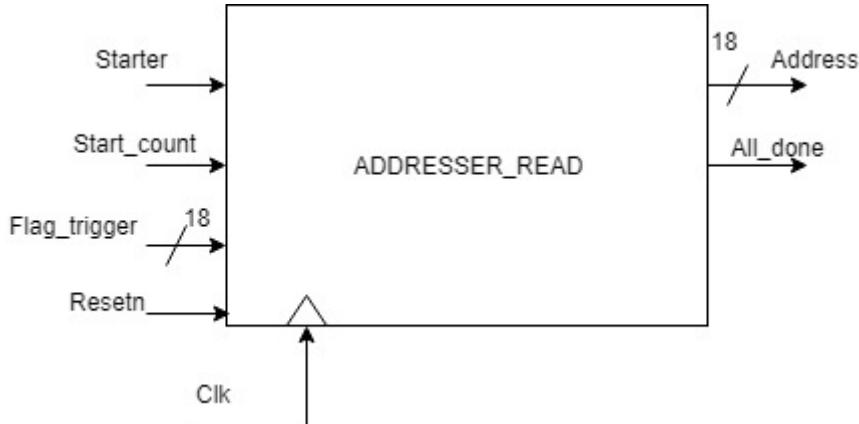
```

```

-- starter<=SW(0);
--resetn<=SW(1);
--CLOCK<=CLOCK_50;
--all_done_record<=SW(2);
--trigger_event<=SW(3);
--LEDG(7 DOWNTO 1)<=address(17 DOWNTO 11);
--LEDG(0)<=All_Done;
--LEDR(17 DOWNTO 0)<=Flag_trigger;
END behavior;

```

Addresser read



Blocco dell'addresser read

L'address_read fornisce il corretto indirizzo di lettura per la sRam.

Un counter è settato al valore di Flag_trigger. Dopo il segnale di start il counter aggiorna il proprio valore ponendosi al primo indirizzo del “passato”. Ad ogni Start_count viene fornito un nuovo indirizzo su Address.

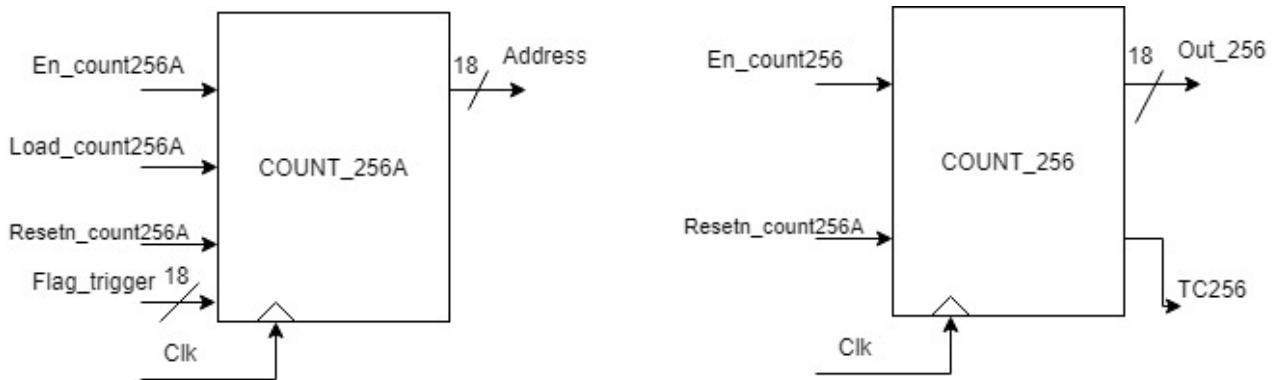
Un secondo counter segnala quando sono stati inviati tutti i 256 000 indirizzi.

Pseudocodice

```
Count=flag_trigger;  
  
If starter=1  
  
    Count=count+1;  
  
    Address=count;  
  
    If Start_count=1  
  
        Count++;  
  
        Count++;  
  
        Address=Count;  
  
        If Count2=256000  
  
            All_done=1;  
  
        Endif;  
  
    Endif;  
  
Endif;
```

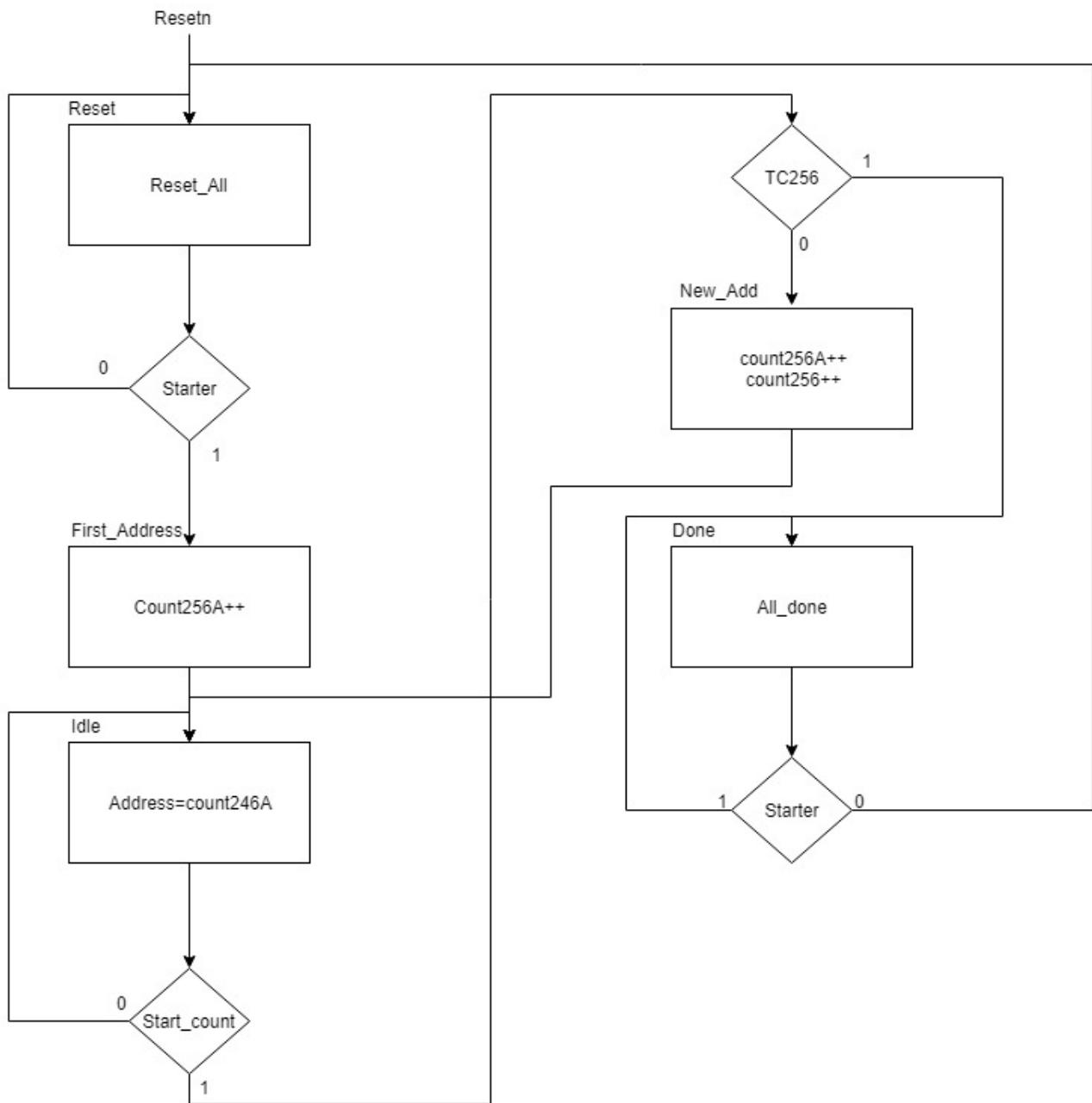
Datapath

```
//DATAPATH
```



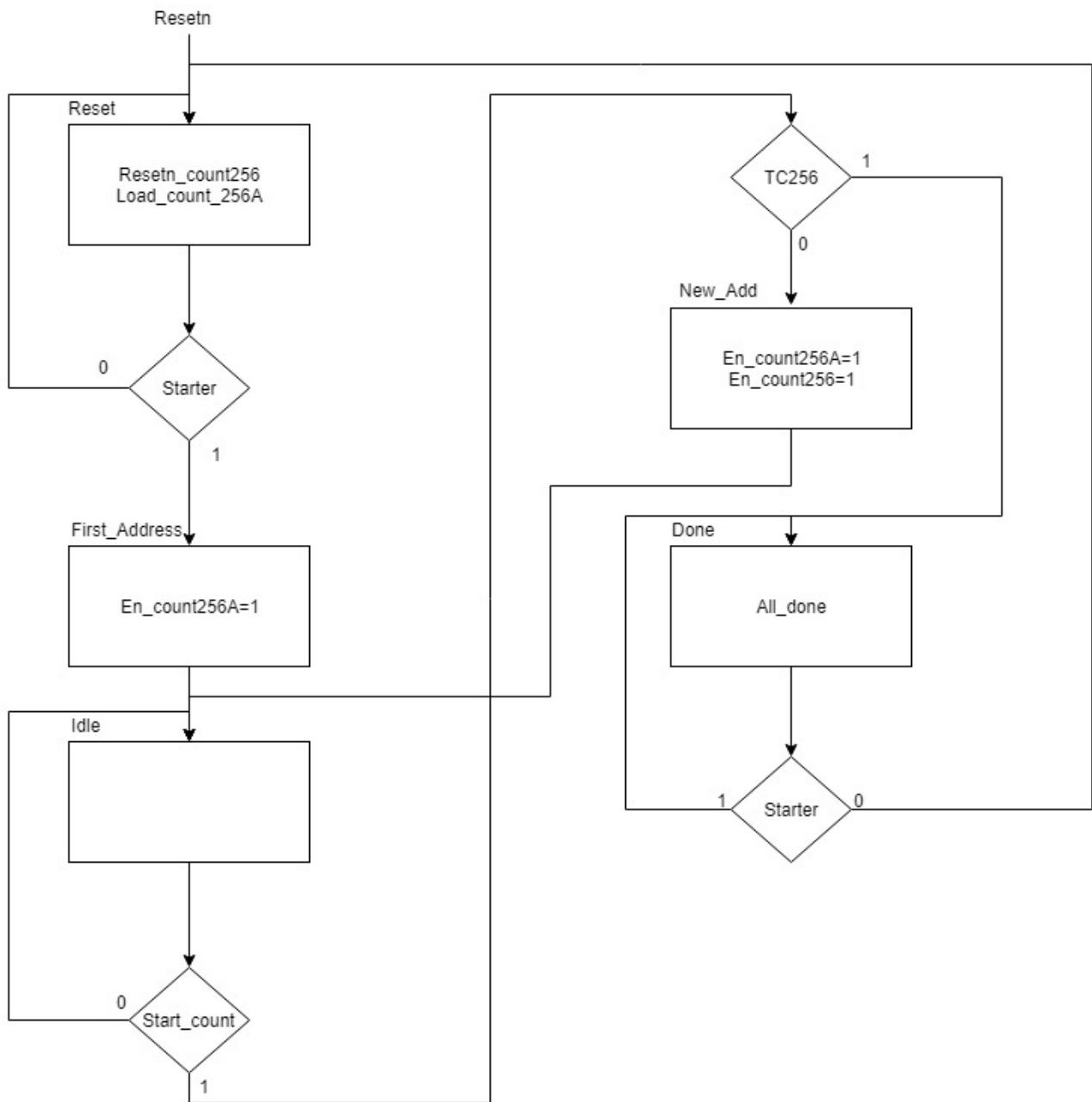
Datapath dell'addresser read

ASM chart



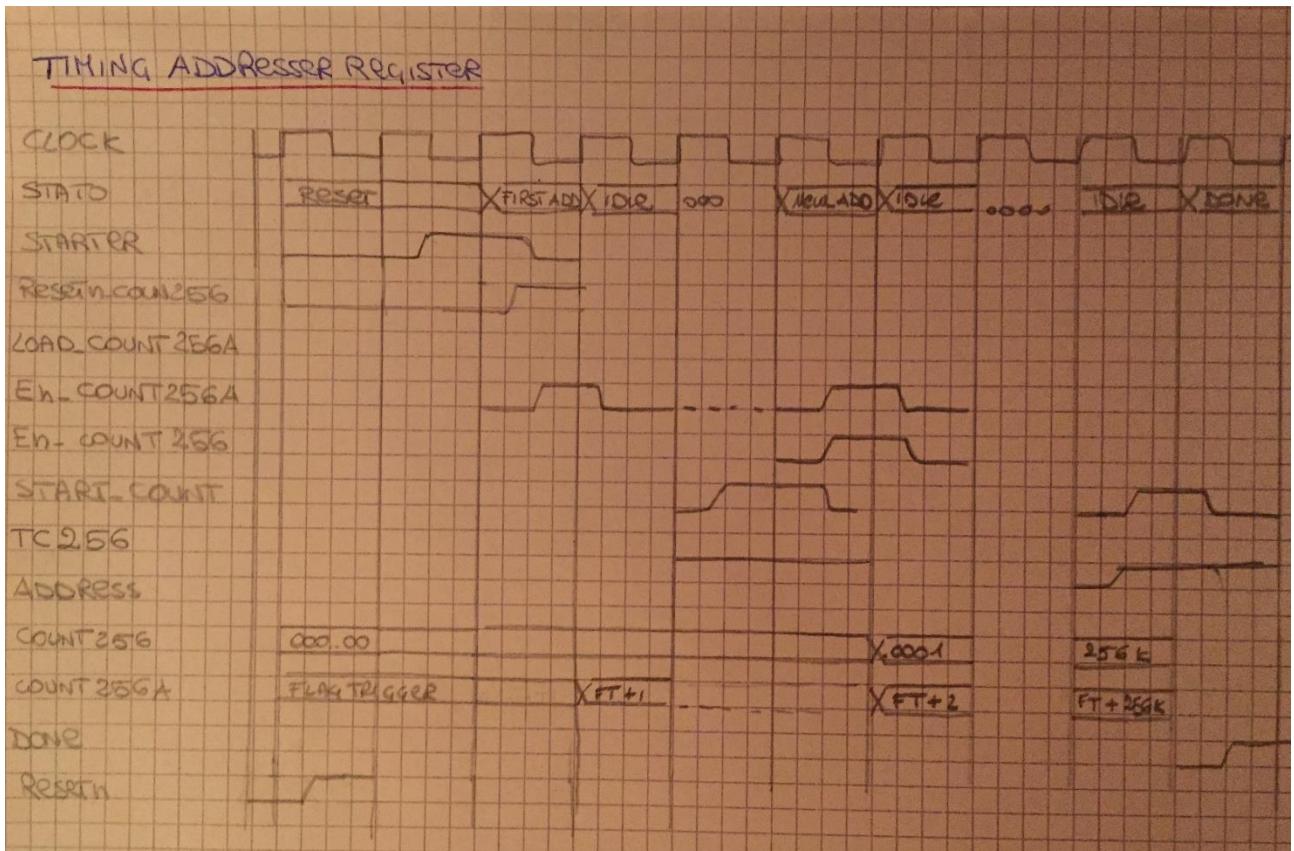
ASM chart dell'addresser read

Control ASM chart



ASM chart dei controlli dell'addresser read

Timing



Timing dell'addresser read

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: addresser_read.vhdl

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY Addresser_read IS
PORT( Starter,Start_count: IN STD_LOGIC;
      Clock,Resetn:IN STD_LOGIC;
      Flag_trigger: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
      Address: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      All_Done: OUT STD_LOGIC);
END Addresser_read;
--PORT( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
--      KEY:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
--      CLOCK_50: IN STD_LOGIC;
--      LEDR: OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
--      LEDG: OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
--END Addresser_read;

```

```

ARCHITECTURE behavior OF Addresser_read IS
--SIGNAL
--SIGNAL starter,Start_count, clock,resetn,All_done: std_logic;
--SIGNAL flag_trigger,Address: std_logic_vector(17 downto 0);
TYPE State_type IS (Reset,First_address,Idle,new_add,Done);
SIGNAL y : State_type;
SIGNAL TC256,en_count256A,load_count256A,resetn_count256A,en_count256,resetn_count256:
STD_LOGIC;
SIGNAL out_256 : std_logic_vector(17 downto 0);
--COMPONENT
COMPONENT counterupN IS
GENERIC (N : integer:=4);
PORT(    Enable : IN STD_LOGIC;
Clock , Resetn : IN STD_LOGIC;
Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;
```

```

component counterupNwithLoad is
GENERIC (N : integer:=10);
port (
cout :out std_logic_vector (N-1 downto 0); -- Output of the counter
data :in std_logic_vector (N-1 downto 0); -- Parallel load for the counter
load :in std_logic; -- Parallel load enable
enable :in std_logic; -- Enable counting
clk :in std_logic; -- Input clock
resetn :in std_logic -- Input reset
);
end component;
```

BEGIN

```

FSM_transitions:
PROCESS ( Clock , Resetn )
```

BEGIN

```

IF Resetn = '0' THEN
y <= Reset;
ELSIF (Clock 'EVENT AND Clock = '1') THEN
CASE y IS
WHEN Reset => IF starter = '1' THEN y <= First_address ;
ELSE y <= Reset ;
END IF ;
```

```

WHEN First_address=>y<=Idle;
WHEN Idle=> IF start_count = '1' THEN IF TC256='1' THEN y <= DONE ;
ELSE y <= New_add ;
END IF ;
ELSE y<=Idle;
```

```

        END IF;
WHEN New_add =>y<=Idle;
WHEN Done => IF Starter = '1' THEN y <= Done ;
    ELSE y <= Reset ;
END IF ;
END CASE;
END IF;
END PROCESS;

```

FSM_outputs: PROCESS(y)

```

BEGIN
All_Done<='0';
Resetn_count256<='1';
en_count256<='0';
en_count256A<='0';
Resetn_count256A<='1';
Load_count256A<='0';

```

CASE y IS

```

WHEN Reset=> Load_count256A<='1';
    Resetn_count256<='0';
WHEN First_address => En_count256A <= '1';
WHEN Idle => Resetn_count256<='1';
WHEN New_add =>en_count256<='1';
    en_count256A<='1';
WHEN Done => All_Done <= '1';
END CASE;
END PROCESS;

```

Counter_256A:counterupNwithLoad

```

GENERIC MAP (N => 18)
PORT MAP (Address,flag_trigger,load_count256A,en_count256A , Clock , Resetn_count256A);

```

Counter_256:counterupN

```

GENERIC MAP (N => 18)
PORT MAP (en_count256 , Clock , Resetn_count256 ,out_256);
TC256 <= out_256(0) AND out_256(1) AND out_256(2) AND out_256(3) AND out_256(4) AND
out_256(5) AND out_256(6) AND out_256(7) AND out_256(8) AND out_256(9) AND out_256(10) AND
NOT out_256(11) AND NOT out_256(12) AND out_256(13) AND out_256(14) AND out_256(15) AND
out_256(16) AND out_256(17);

```

```

-- starter<=KEY(1);
-- resetn<=KEY(0);
-- CLOCK<=CLOCK_50;
-- FLAG_TRIGGER<=SW(17 downto 0);
-- LEDR<=Address;
-- LEDG(0)<=All_Done;

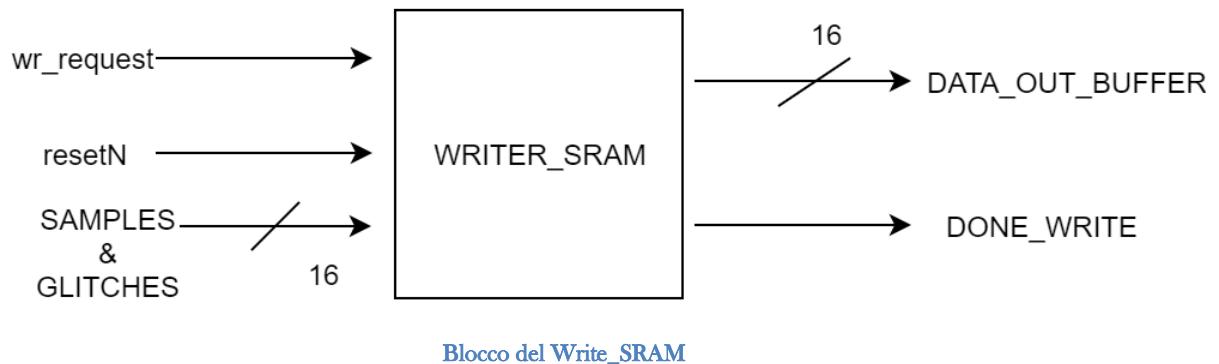
```

END behavior;

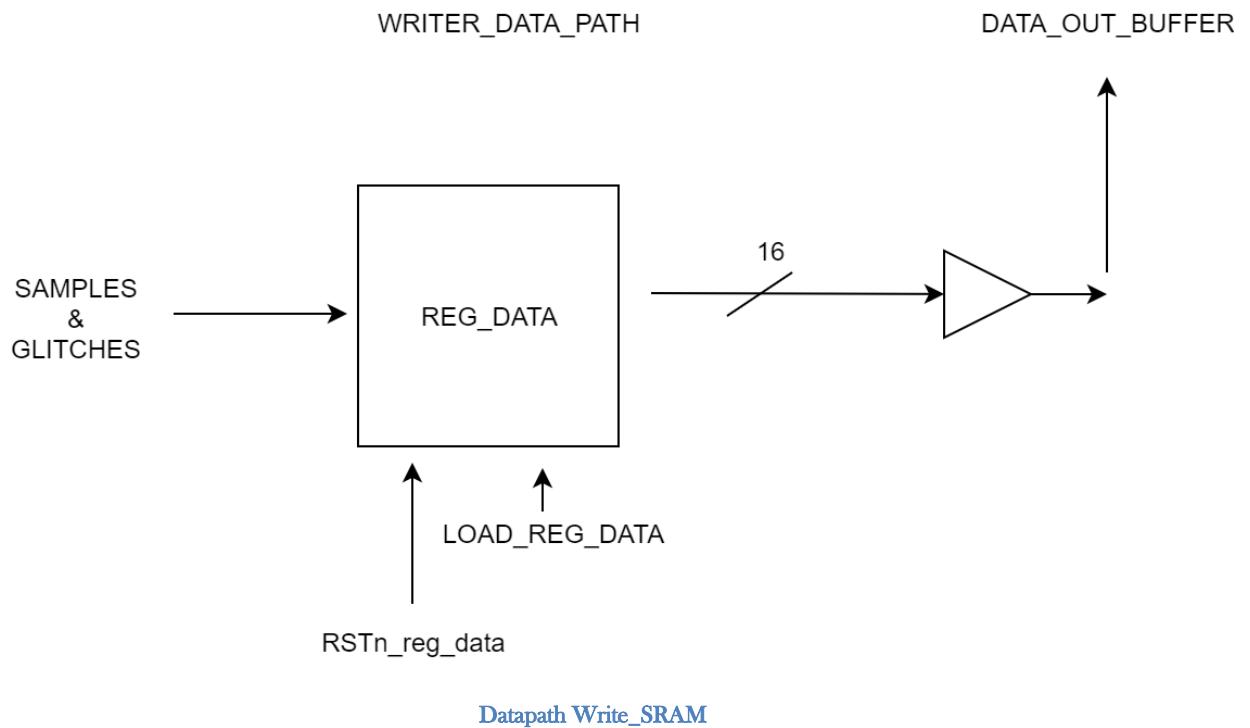
Write_SRAM

Il writer_sram è il blocco che si occupa di dare la giusta temporizzazione del passaggio dei dati tra il resto del circuito e la memoria. Esso gestisce solo il passaggio dei dati poiché il controllo è affidato a un altro blocco hardware il control_sram per evitare dei conflitti sul circuito. Il blocco ha al suo interno dei driver che tramite un segnale di abilitazione permettono o meno il passaggio dei dati (quando scrivo abilito il passaggio, quando leggo non abilito il passaggio dei dati).

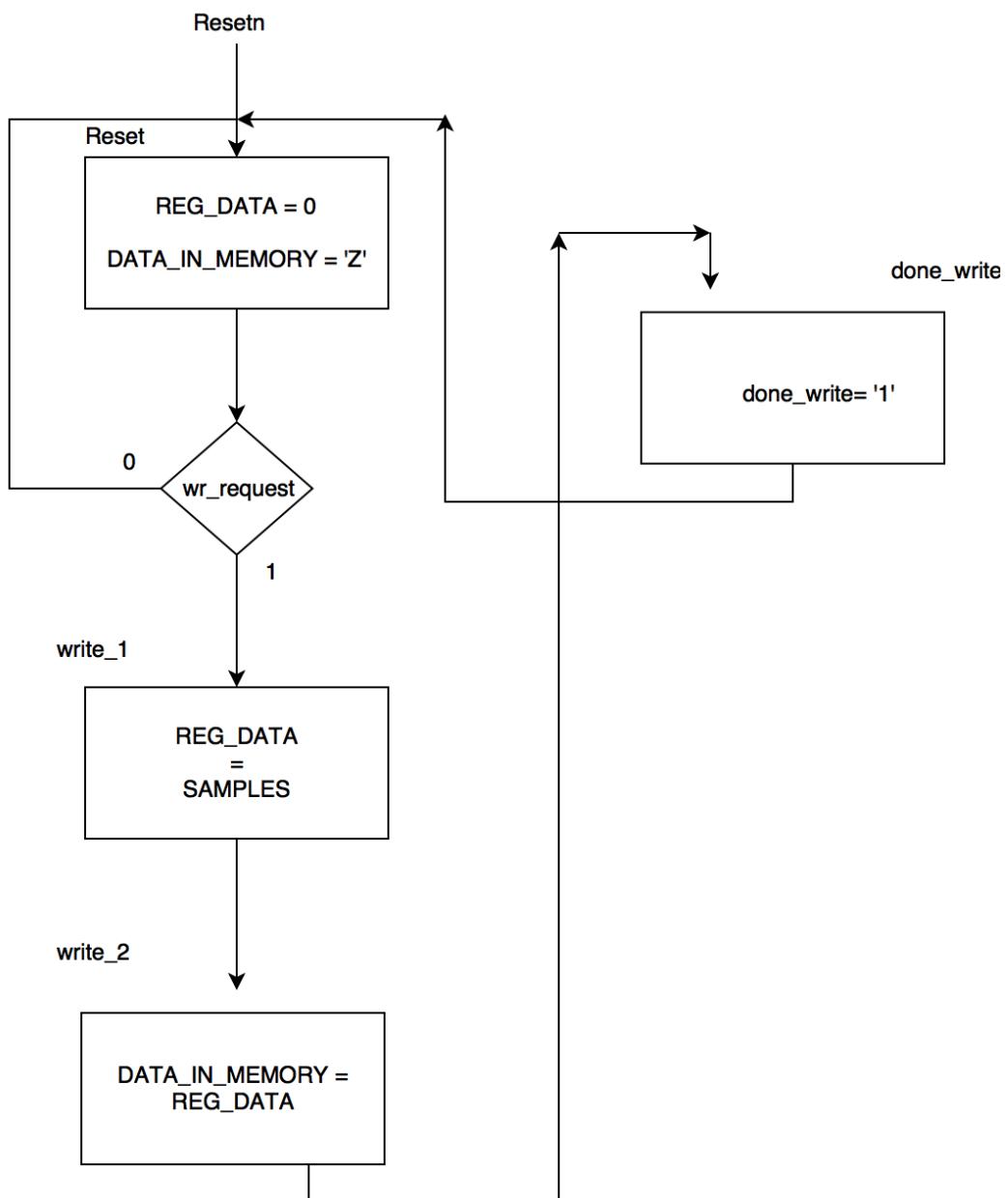
WRITER_SRAM_TOLEVEL



Datapath

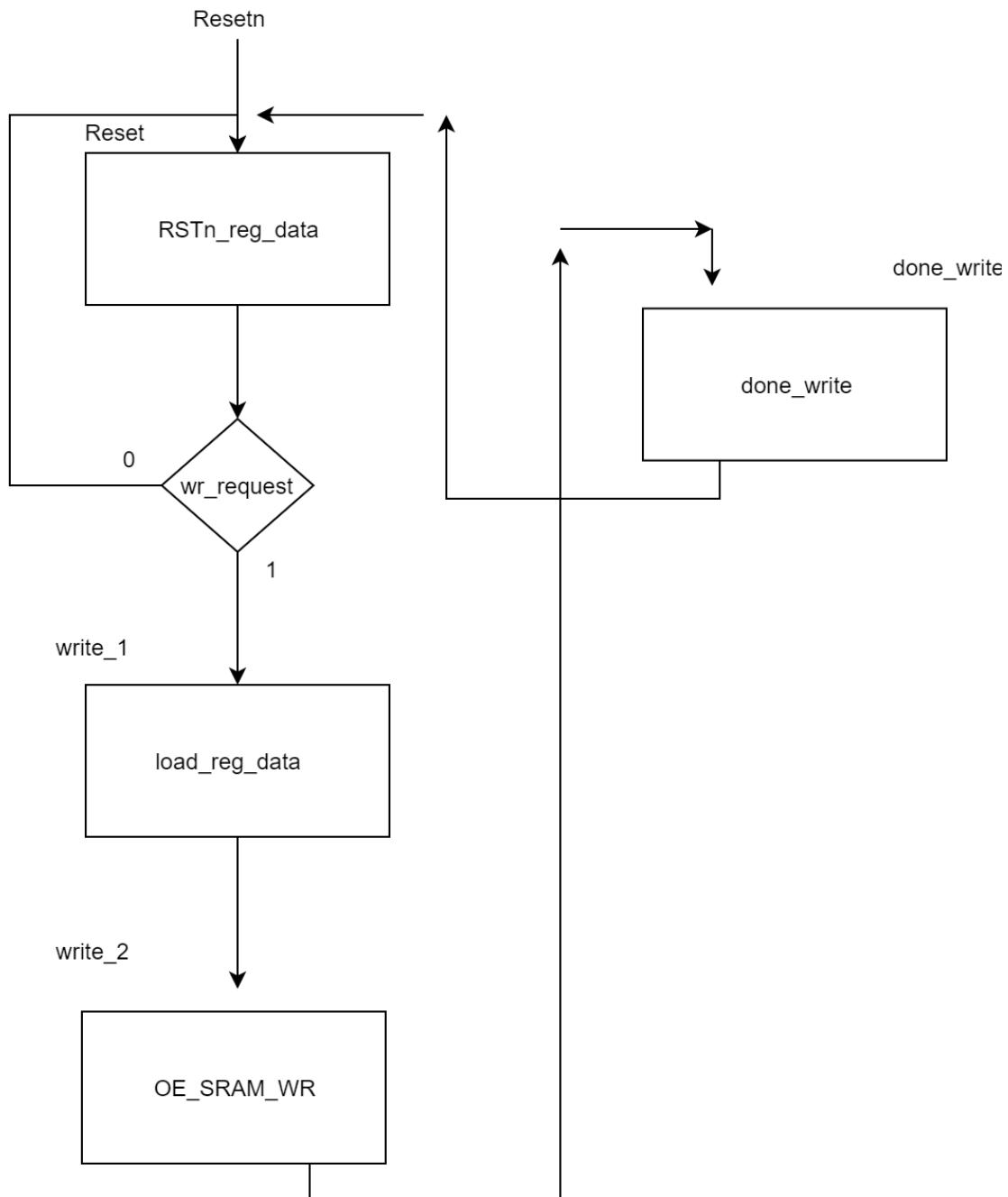


ASM chart



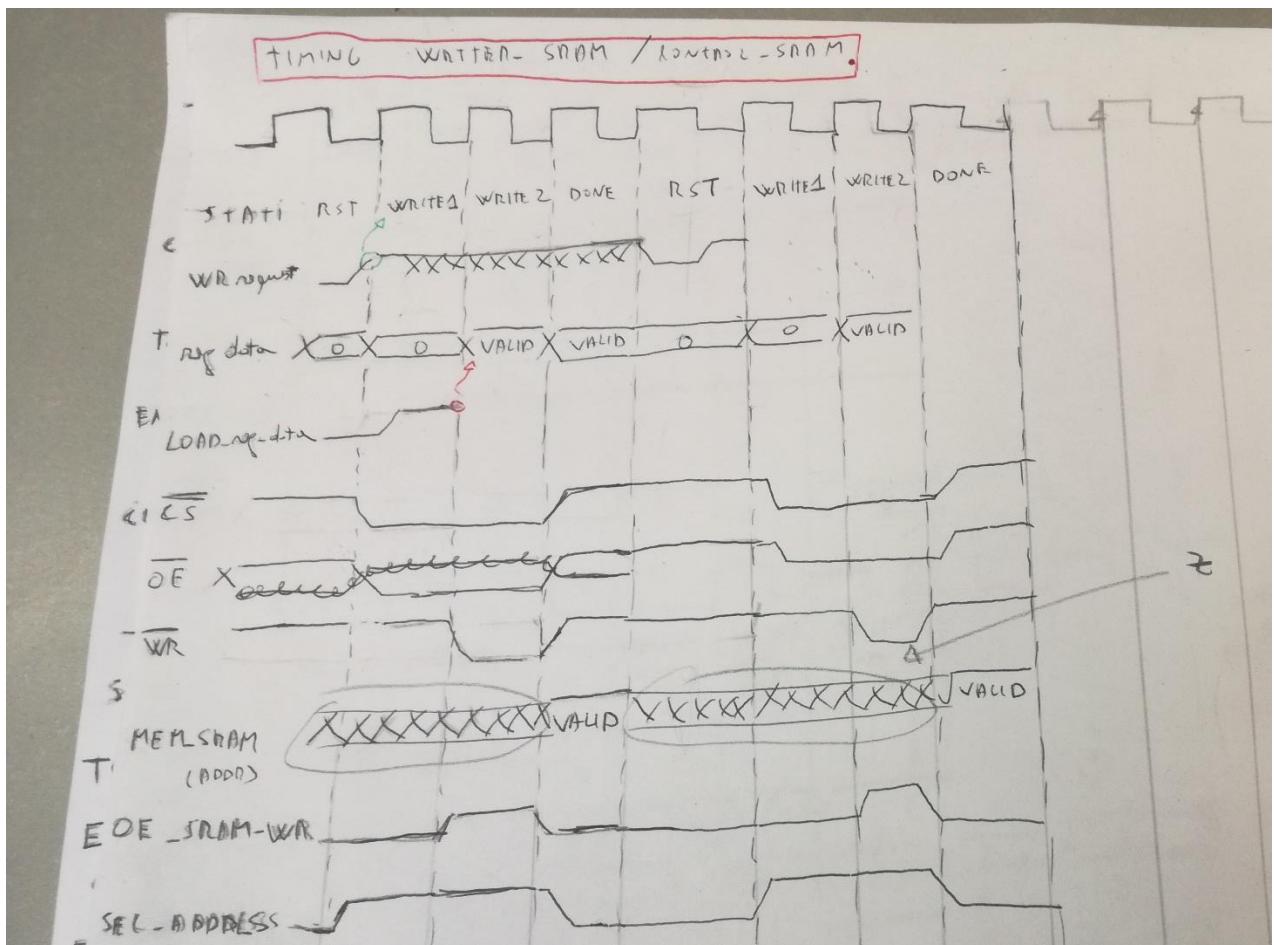
ASM chart del Write_SRAM

Control ASM chart



ASM chart dei controlli del Write_SRAM

Timing



Timing del Write_SRAM

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: writer_SRAM.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

ENTITY writer_sram is

```
PORT ( wr_request : IN STD_LOGIC;
clock , resetN : IN STD_LOGIC;
SAMPLES , GLITCHES : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
DATA_OUT_BUFFER : OUT STD_LOGIC_VECTOR ( 15 DOWNTO 0);
DONE_WRITE : OUT STD_LOGIC);
```

```
END writer_sram;
```

ARCHITECTURE BEHAVIOUR OF writer_sram IS

--component

```
component reg16 IS
```

```
PORT ( D : IN STD_LOGIC_VECTOR(15 DOWNTO 0) ;  
Resetn, Clock, load : IN STD_LOGIC ;  
Q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
```

```
END component ;
```

```
component tri_16 IS
```

```
PORT ( X : IN STD_LOGIC_VECTOR(15 DOWNTO 0);  
E : IN STD_LOGIC;  
F : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
```

```
END component;
```

```
TYPE state_type is (RESET_write, write_1 , write_2 , done_writer);
```

```
SIGNAL STATE : state_type;
```

--signal interni

```
signal OE_SRAM_WR : STD_LOGIC;
```

```
signal rstn_reg_data : std_logic;
```

```
signal load_reldata : std_logic;
```

```
signal data , data_out : std_logic_vector ( 15 downto 0);
```

```
begin
```

```
FSM_transitions: PROCESS ( resetN , clock )
```

```
BEGIN
```

```
IF (resetN = '0') THEN
```

```

STATE <=RESET_write;

ELSIF (Clock'EVENT AND Clock = '1') THEN

CASE STATE IS

WHEN reset_write => IF wr_request = '1' THEN STATE<= write_1 ; ELSE STATE <= RESET_write ;
END IF ;

WHEN write_1 => state <= write_2;

WHEN write_2 => state <= done_writer;

when done_writer => state <= reset_write ;

WHEN OTHERS => STATE <= reset_write;           -- controllo

END CASE ;
END IF ;
END PROCESS ;

```

```

FSM_OUTPUT : PROCESS (STATE) IS

BEGIN

OE_SRAM_WR <= '0';

DONE_WRITE <= '0'; rstn_reg_data <= '1'; load_regdata <= '0';
OE_SRAM_WR <= '0';

CASE STATE is

WHEN RESET_write => rstn_reg_data <= '0'; -OE_SRAM_WR <= '0';

WHEN write_1 => load_regdata <= '1'; -OE_SRAM_WR <= '1';

WHEN write_2 => OE_SRAM_WR <= '1'; -WR_SRAM <= '1'; CS_SRAM <= '1';

when done_writer => DONE_WRITE <= '1';

--WHEN wait_data => enable_count <= '1';

END CASE;

```

```

END PROCESS;

```

```

data <= samples & glitches;

reg_data : reg16 port map ( data, rstn_reg_data , clock , load_regdata , data_out);

tri_state : tri_16 port map ( data_out , OE_SRAM_WR , DATA_OUT_BUFFER);

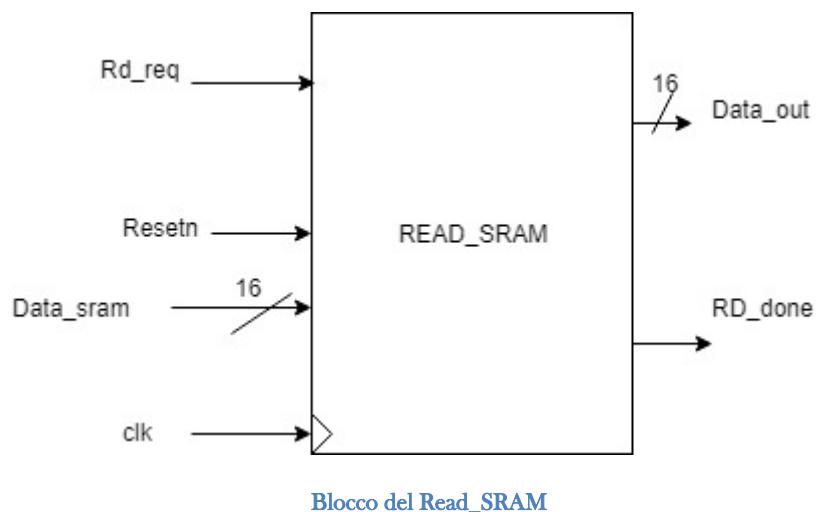
end behaviour;

```

Read_SRAM

Il blocco si occupa di memorizzare i dati in arrivo dalla SRAM.

Il blocco è sintetizzabile come sotto.



Pseudocodice

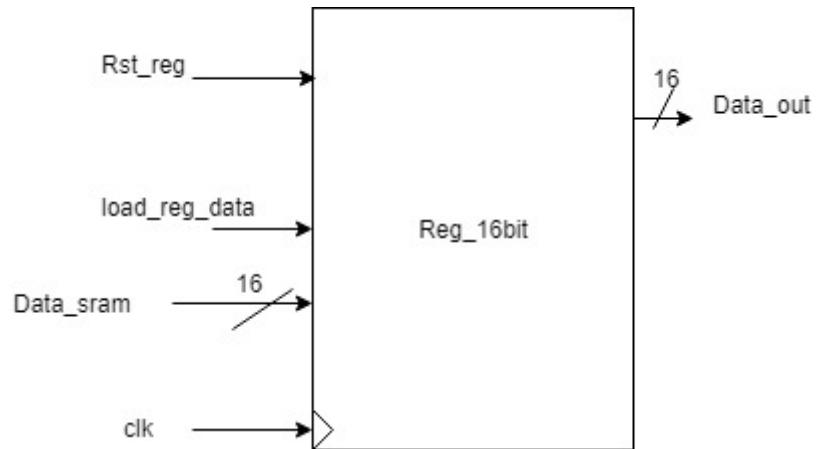
```

RESET REG16
BUFFER HI-Z
IF RD_REQ = 1
    LOAD REG16
    CNT_STATE = 0
    (colpo di clock successivo)

```

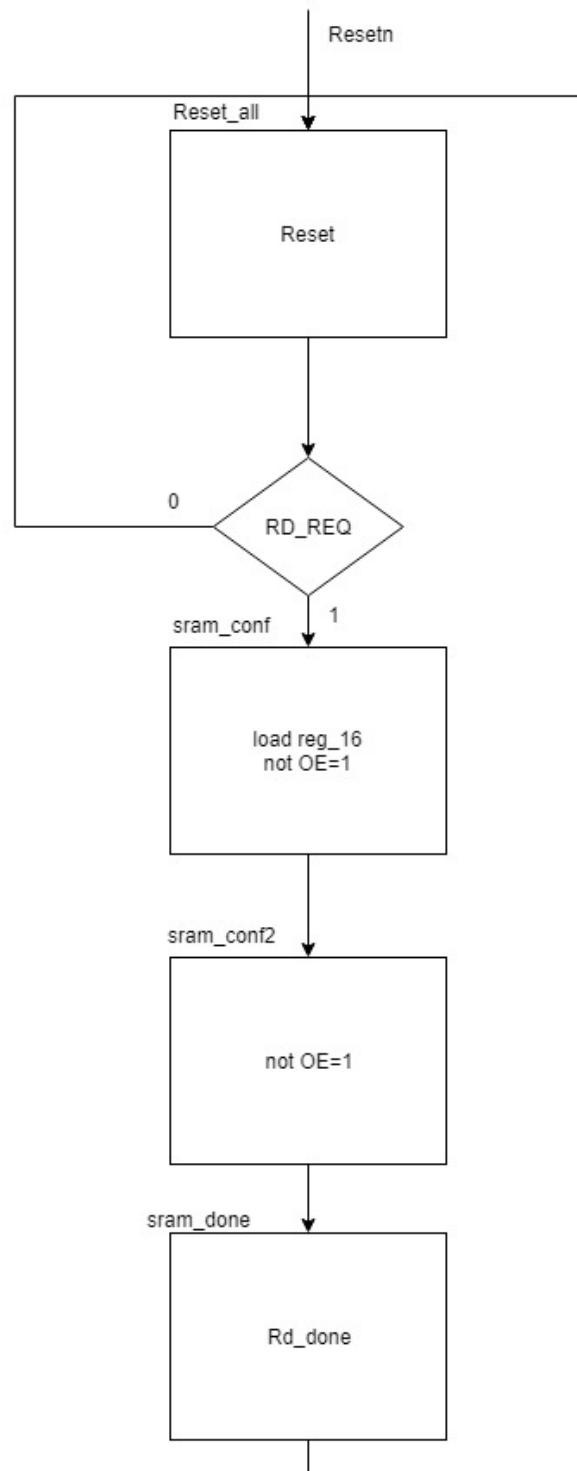
```
CNT_STATE = 1  
(colpo di clock successivo)  
RD_DONE = 1  
END IF;
```

Datapath



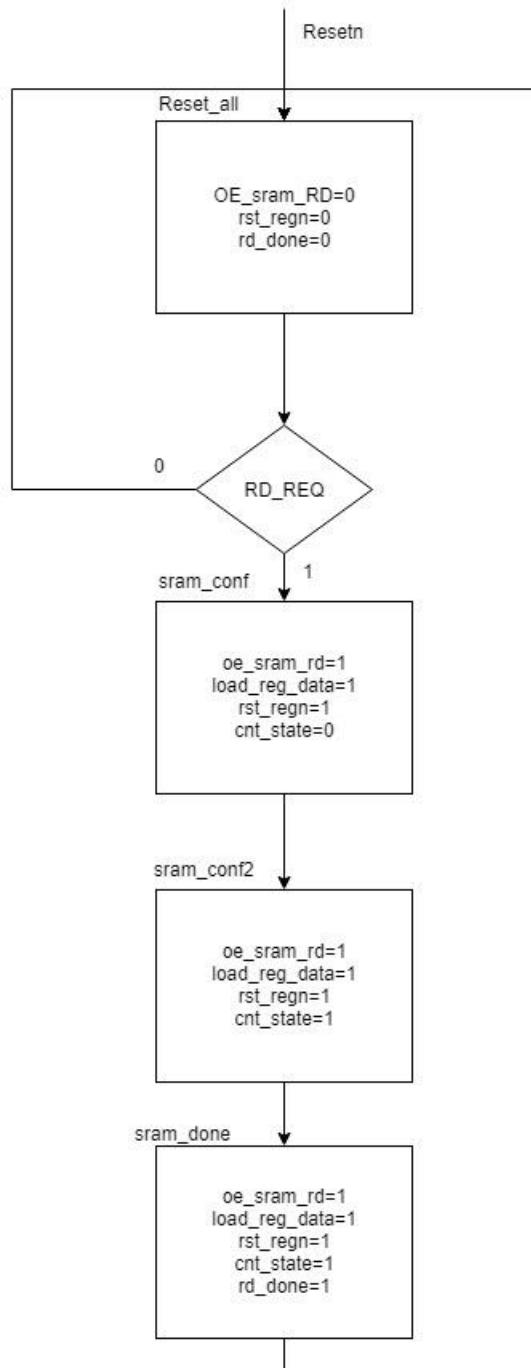
Datapath del Read_SRAM

ASM chart



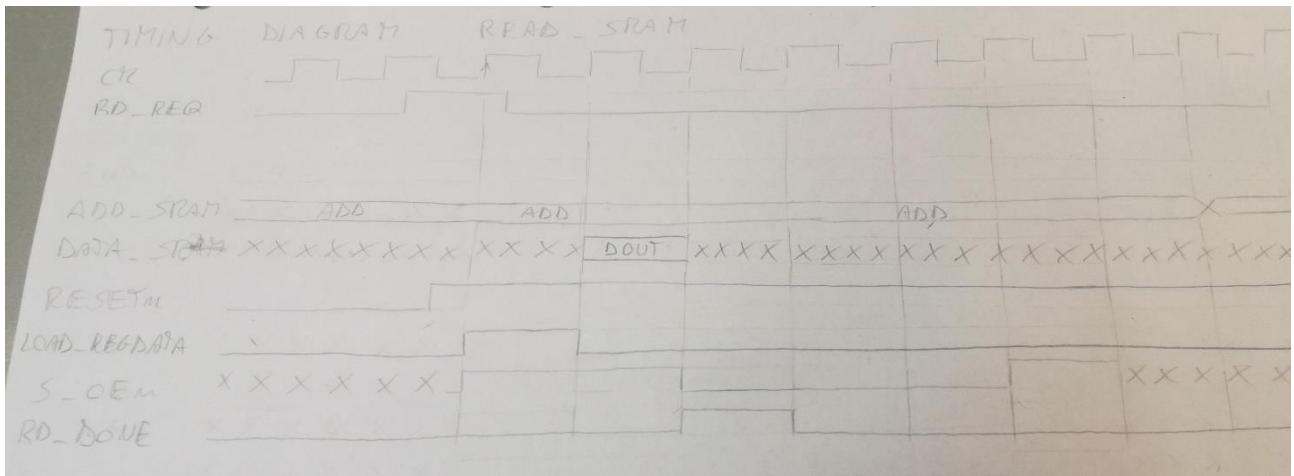
ASM chart del Read_SRAM

Control ASM chart



ASM chart dei controlli del Read_SRAM

Timing



Timing del Read_SRAM

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: read_SRAM.vhdl

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY Read_SRAM IS
PORT(
    -input
    RD_REQ: IN STD_LOGIC;
    Clock, Resetn: IN STD_LOGIC;
    Data_SRAM: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    -output
    Dout: OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    RD_Done: OUT STD_LOGIC);
END Read_SRAM;
--Example to implement this on FPGA
--PORT( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
--      KEY:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
--      LEDG:OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
--      LEDR: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
--END Ascii_transmission ;

```

```

ARCHITECTURE behavior OF Read_SRAM IS
--SIGNAL
--SIGNAL starter,clock,resetn,error,error_data,All_done_T,Full_out,All_done: std_logic;
--signal data_in,glitch_in,data_out: std_logic_vector(7 downto 0);
TYPE State_type IS (Reset_all,SRAM_Conf, SRAM_Conf2, SRAM_Done);
SIGNAL y : State_type;

```

```

SIGNAL Rst_regn: STD_LOGIC;
SIGNAL Load_reg_data: STD_LOGIC;
SIGNAL cnt_state: STD_LOGIC;
SIGNAL OE_SRAM_RD: STD_LOGIC;
SIGNAL DATA_IN: STD_LOGIC_VECTOR(15 downto 0);

```

-COMPONENT

```

COMPONENT reg16 IS
PORT ( D : IN STD_LOGIC_VECTOR(15 DOWNTO 0) ;
Resetn, load : in std_logic;
clock : IN STD_LOGIC;
Q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END COMPONENT;

```

BEGIN

FSM_transitions:
PROCESS (Clock , Resetn)

BEGIN

```

IF Resetn = '0' THEN
  y <= Reset_all;
ELSIF (Clock 'EVENT AND Clock = '1') THEN
  CASE y IS
    WHEN Reset_all => IF RD_REQ = '0' THEN y <= Reset_all;
    ELSE y <=
SRAM_Conf;
    END IF;
    WHEN SRAM_Conf => y <= SRAM_Conf2;
    WHEN SRAM_Conf2 => y <= SRAM_Done;
    WHEN OTHERS => y <= reset_all;
  END CASE;
END IF;
END PROCESS;

```

FSM_outputs: PROCESS(y)

```

BEGIN
rst_regn <= '0';
Load_reg_data <= '0';
OE_SRAM_RD <= '0';
RD_Done <= '0';
cnt_state <= '0';

```

```

CASE y IS
WHEN Reset_all=>
  OE_SRAM_RD <= '0';
  rst_regn <= '0';

```

```

RD_Done <= '0';
cnt_state <= '0';

WHEN SRAM_Conf =>
    OE_SRAM_RD <= '1';
    rst_regn <= '1';
    Load_reg_data <= '1';
    cnt_state <= '0';

WHEN SRAM_Conf2 =>
    OE_SRAM_RD <= '1';
    rst_regn <= '1';
    Load_reg_data <= '1';
    cnt_state <= '1';

WHEN SRAM_Done =>
    OE_SRAM_RD <= '1';
    rst_regn <= '1';
    Load_reg_data <= '1';
    cnt_state <= '1';
    RD_Done <= '1';

END CASE;
END PROCESS;

```

--DATAPATH

```

--register 16bit
reg_16bit : reg16 port map (data_in, rst_regn, load_reg_data, Clock , Dout);

END behavior;

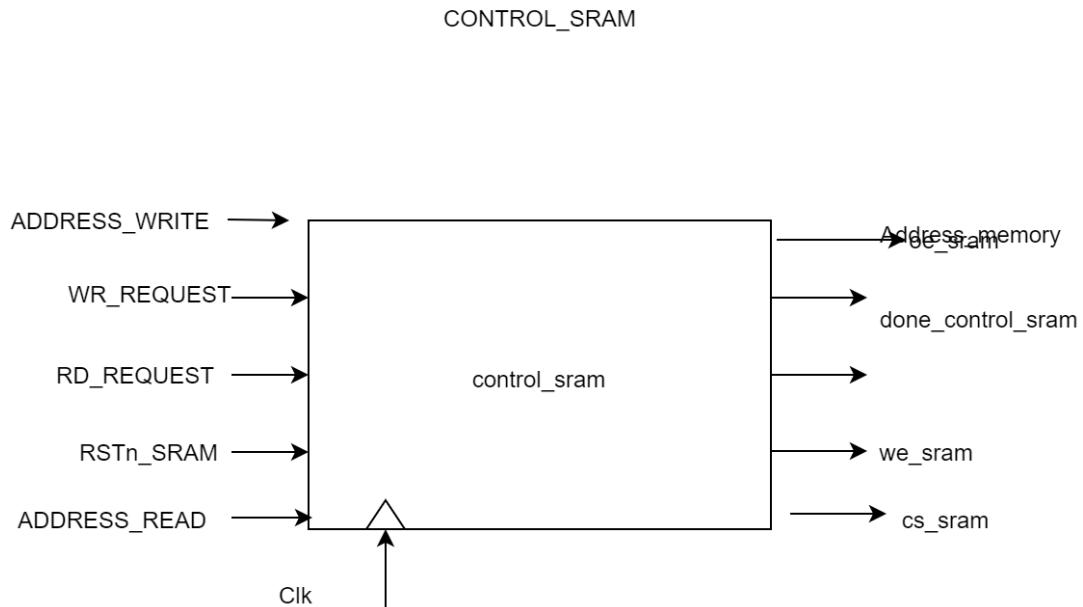
```

Control_SRAM

Il blocco hardware si deve interfacciare con la memoria e mandare i segnali alla memoria con la giusta temporizzazione (data dal datasheet), deve avere un timing “parallelo” a quello del blocco writer_sram e read_sram in modo da avere i dati disponibili e comunicare con la memoria per scrivere o leggere.

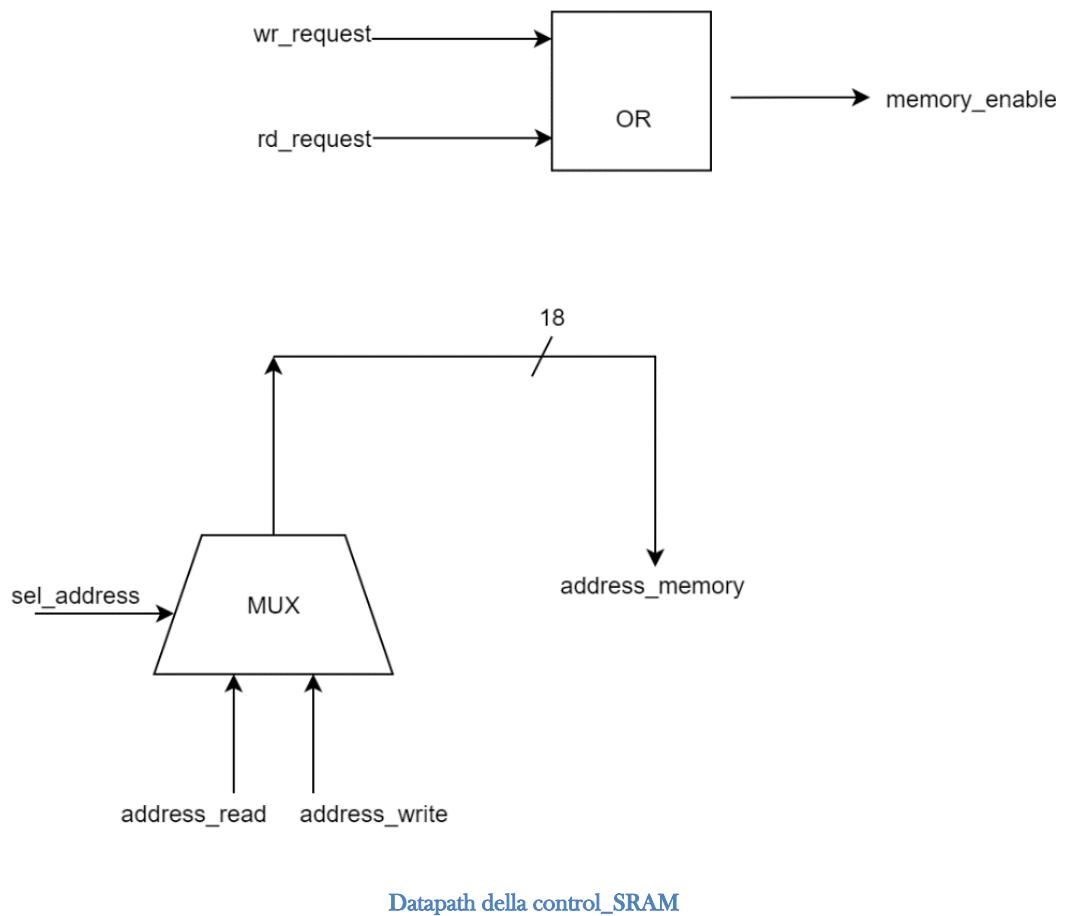
L'indirizzo arriva dall' address_sram e la control_sram deciderà in maniera intelligente se mandare alla memoria l'indirizzo da leggere o scrivere.

La control_sram tramite la lettura dei segnali di rd_request e wr_request decide intelligentemente che percorso intraprendere tra la lettura e la scrittura (che sono stati progettati in modo parallelo ovvero con macchine con lo stesso numero di stati) mandando i segnali corretti alla memory per l'interfacciamento.

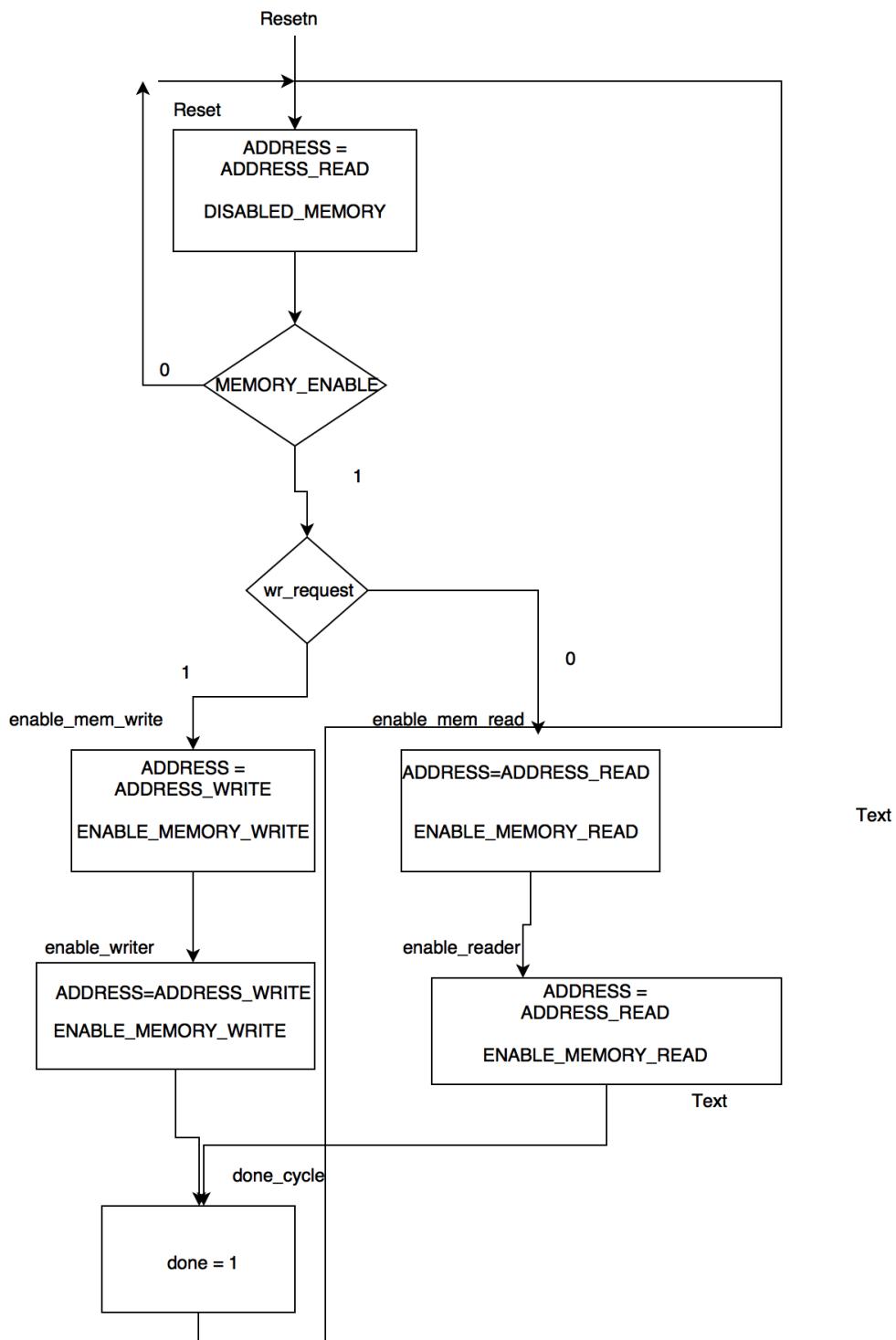


Blocco della control_SRAM

Datapath

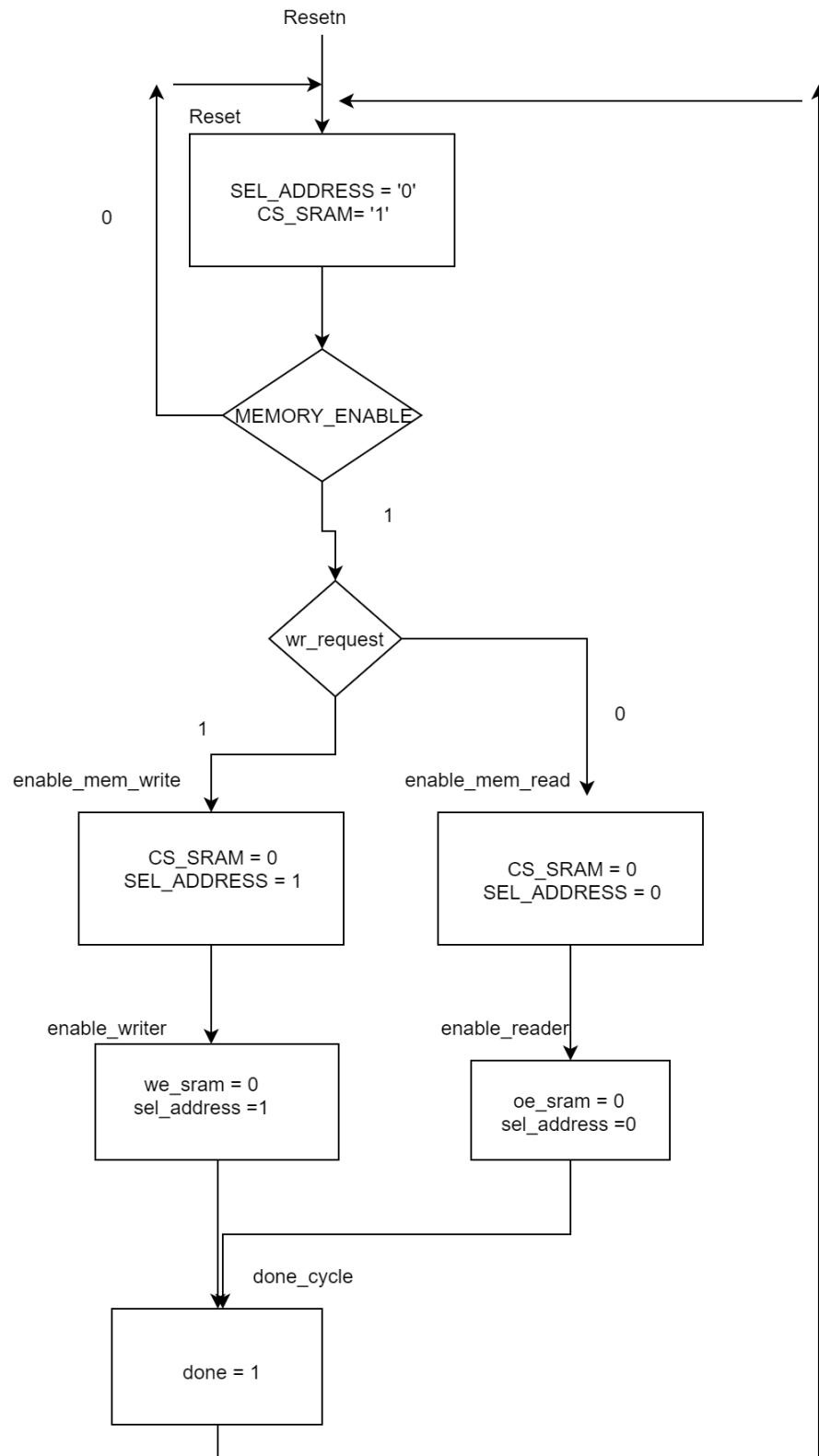


ASM chart



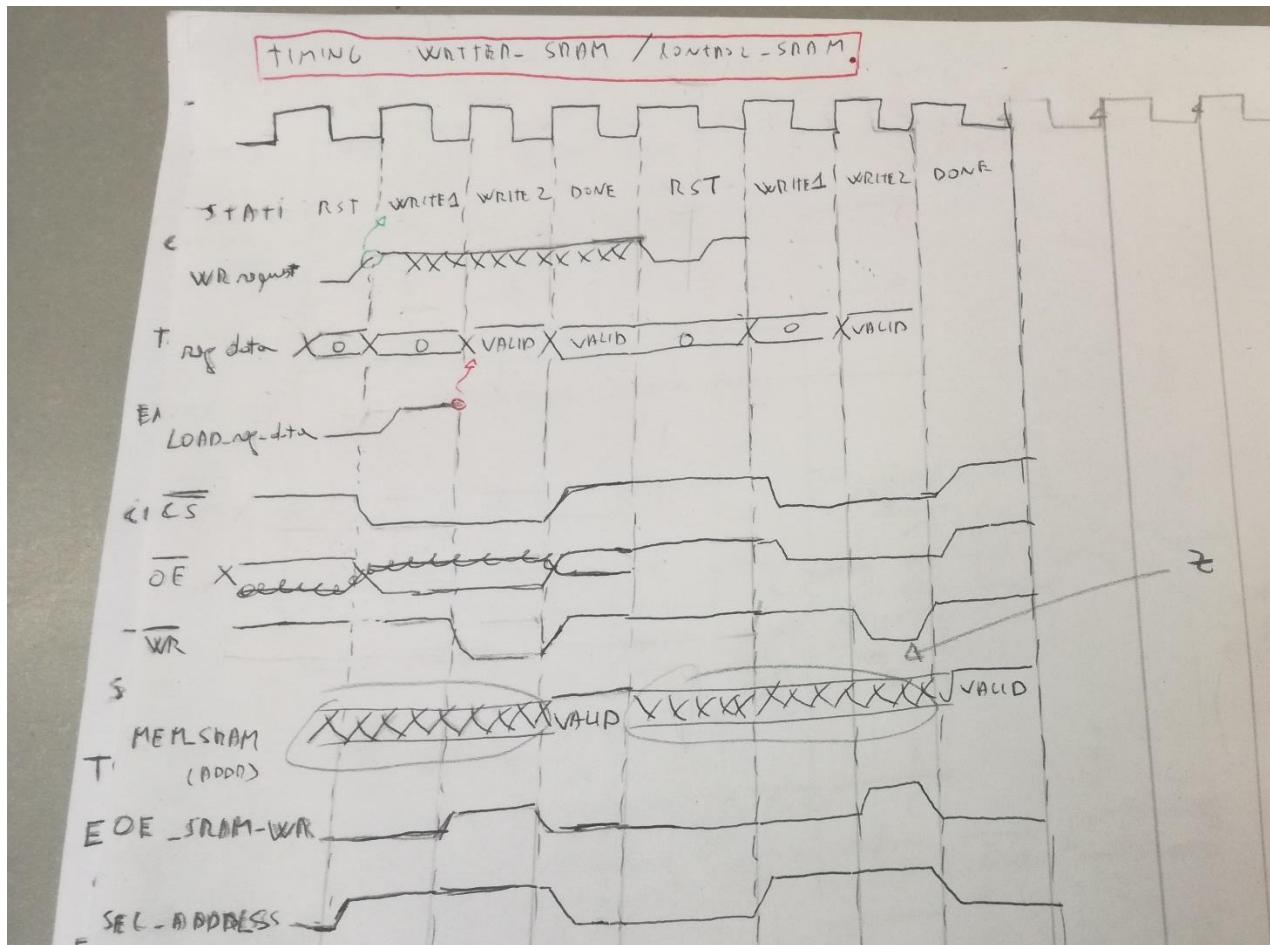
ASM CHART della control_SRAM

Control ASM chart



ASM dei controlli della control_SRAM

Timing



Timing della control_SRAM

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: control_SRAM.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
entity control_sram is
```

```
port ( wr_request , rd_request , rstn_sram , clock : in std_logic;
      address_read , address_write : in std_logic_vector( 17 downto 0);
      address_memory : out std_logic_vector ( 17 downto 0 );
```

```

done_control_sram : out std_logic;

cs_sram , we_sram , oe_sram : out std_logic);

end control_sram;

```

architecture Behaviour of control_sram is

--component

```

component mux2to1 IS
PORT ( w0, w1 : IN STD_LOGIC_vector( 17 downto 0);
f : OUT STD_LOGIC_vector(17 downto 0);
s : in std_Logic);

```

END component;

TYPE state_type is (RESET_sram , enable_memory_read , enable_memory_write , enable_writer,
enable_reader, done_cycle);

--signal

```

SIGNAL STATE : state_type;

signal sel_address, memory_enable : std_logic;

```

begin

FSM_transitions: PROCESS (rstn_sram , clock)

BEGIN

IF (rstn_sram = '0') THEN

STATE <=RESET_sram; -RESET ASINCRONO

ELSIF (Clock'EVENT AND Clock = '1') THEN

CASE STATE IS

WHEN RESET_sram => if memory_enable = '1' then if wr_request = '1' then state <= enable_memory_write;
else state <= enable_memory_read; end if ; else state <=reset_sram; end if;

```

WHEN enable_memory_write => state <= enable_writer;

WHEN enable_memory_read => state <= enable_reader;

WHEN enable_writer => STATE <= DONE_CYCLE;

when enable_reader => STATE <= DONE_CYCLE;

when done_cycle => STATE <= RESET_SRAM;

WHEN OTHERS => STATE <= RESET_sram;           -- controllo

END CASE;
END IF;
END PROCESS ;

```

FSM_OUTPUT : PROCESS (STATE) IS

BEGIN

```
cs_sram <= '1'; we_sram <= '1' ; oe_sram <= '1'; done_control_sram <= '0'; sel_address <= '0' ;
```

CASE STATE is

```
WHEN RESET_sram => sel_address <= '0'; cs_sram <= '1' ;
```

```
WHEN enable_memory_write => cs_sram <= '0'; sel_address <= '1';           --we_sram <= '0';
```

```
WHEN enable_memory_read => cs_sram <= '0'; sel_address <= '0';           --we_sram <= '1';
```

```
WHEN enable_writer => we_sram <= '0'; sel_address <='1' ;
```

```
when enable_reader => oe_sram <= '0'; sel_address <='0' ;
```

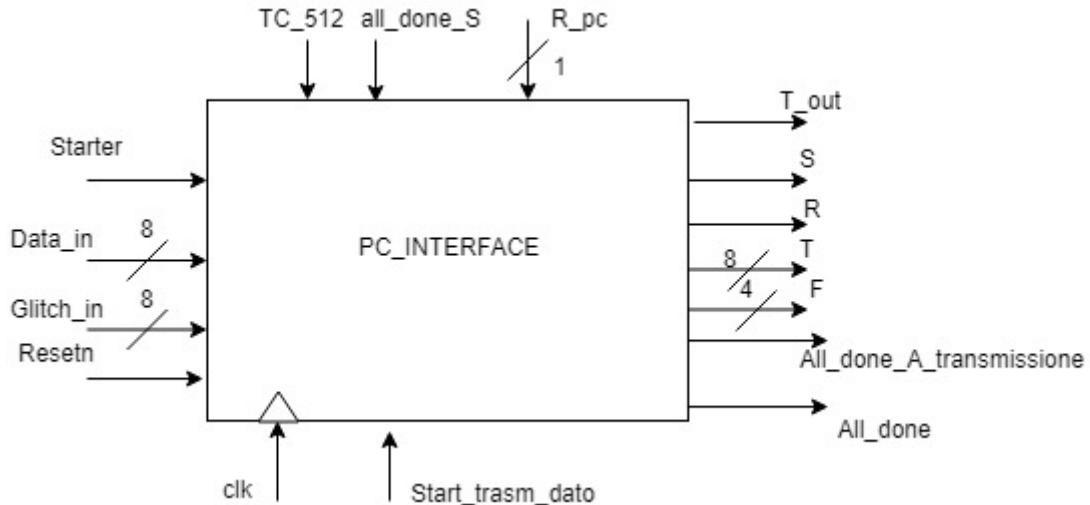
```
WHEN done_cycle => done_control_sram <= '1';
```

```
END CASE;
END PROCESS;
```

--DATAPATH

```
mux_addresser : mux2to1 port map (    address_read , address_write , address_memory , sel_address );  
memory_enable <= wr_request or rd_request;  
END Behaviour;
```

PC interface



Blocco della PC interface

Pc_interface è un blocco che permette di interfacciare l'analizzatore al Pc. E' composto da un blocco UART per la ricezione e la trasmissione dei dati in codice ascii. Un blocco chiamato Ascii_Translator traduce i dati ricevuti da codice Ascii a binario. Il blocco Ascii_Transmission, infine, traduce i dati da binario a Ascii prima che vengano inviati.

I dati seriali vengono ricevuti su R_pc. S,R,T_out sono segnali che si attivano nel caso venga ricevuta la lettera corrispondente. Su T in uscita troviamo il valore di Trigger, mentre su F l'informazione sulla frequenza di campionamento.

Su data_in e glitch_in vengono inseriti i dati da inviare al PC.

Start_trasm_dato e all_done_A_transmissione sono segnali gestiti direttamente dalla main FSM.

Tc512 segnala che tutta la memoria è stata inviata come dato in uscita.

Pseudocodice

```

Reg_F=0;
Reg_T=0;
Reg_s=0;
Reg_R=0;
Start_ricezione;
While(done_ricezione=1);
    
```

```

If ack=1 // trasmetti K

Data_in_T=K; start_trasmissione      Macro_T_errore

While(done_trasmissione=1)

Elsif(data_out_R='F')

Start_ricezione;

While(done_ricezione=1)

If ack=1 Macro_T_errore;

Else Data_in_trad_ascii=Data_out_ricez;

start_traduz_ascii;

while(all_done=1)

if(errore=1) Macro_T_errore;

else

reg_F=data_out_traduttore;

endif;

endif;

Elsif(data_out_R=T)

Start_ricezione;

While(done_ricezione=1)

If ack=1 Macro_T_errore;

Else Data_in_trad_ascii=Data_out_ricez;

start_traduz_ascii;

while(all_done=1)

if(errore=1) Macro_T_errore;

else

reg_T(7 downto 4)=data_out_traduttore;

macro_T_o;

end if;

end if;

```

```

Start_ricezione;

While(done_ricezione=1)

    If ack=1 Macro_T_errore;

    Else Data_in_trad_ascii=Data_out_ricez;

        start_traduz_ascii;

        while(all_done=1)

            if(errore=1) Macro_T_errore;

            else

                reg_T(3 downto 0)=data_out_traduttore;

                macro_T_o;

            endif;

        endif;

    elsif(Data_outR=R)

        Reg_R=1; macro_trasm_o;

    elsif(Data_outR=S)

        if Reg_control_T=1

            Reg_S=1; macro_trasm_o;

        else

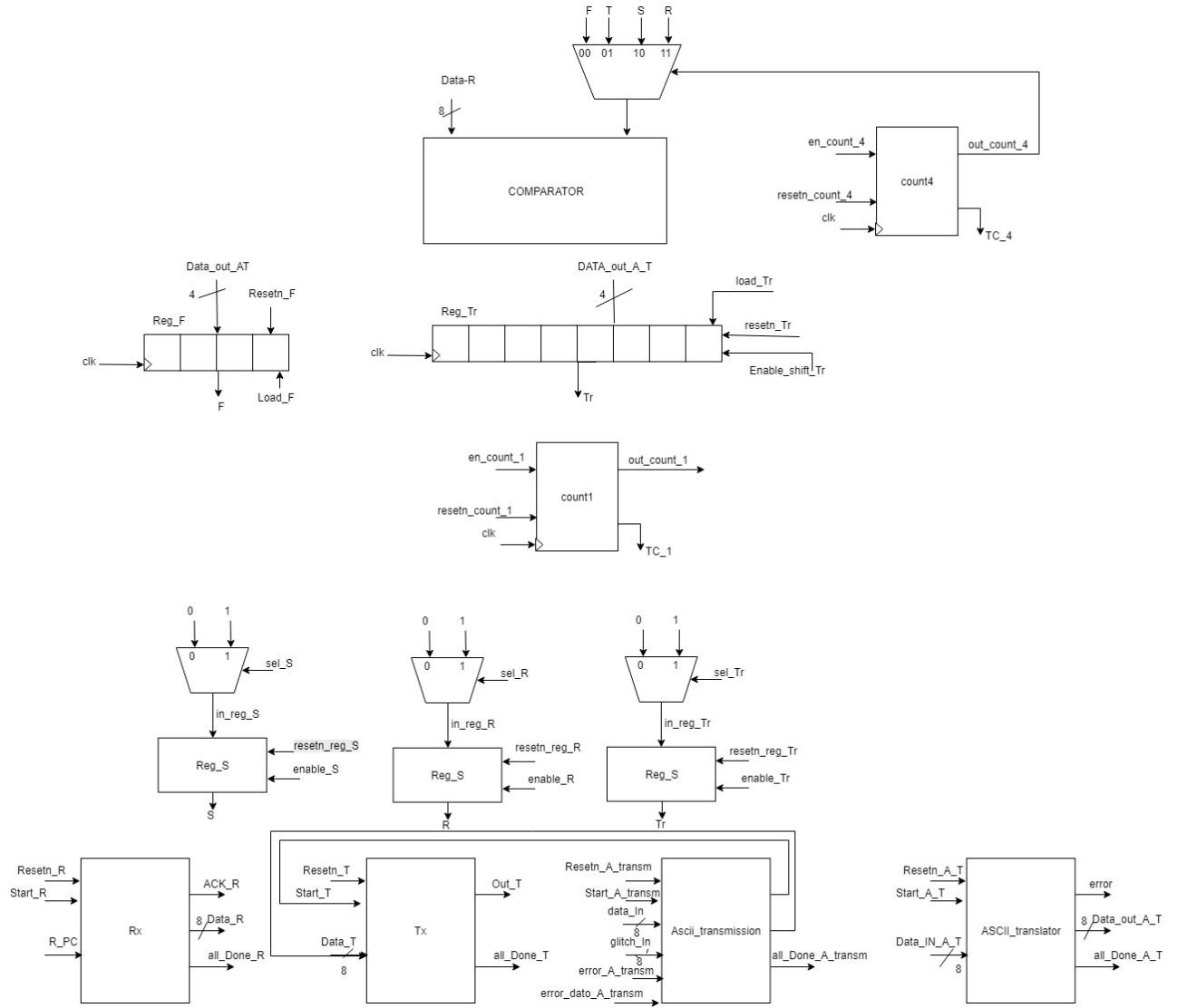
            macro_T_error;

        endif;

    endif;

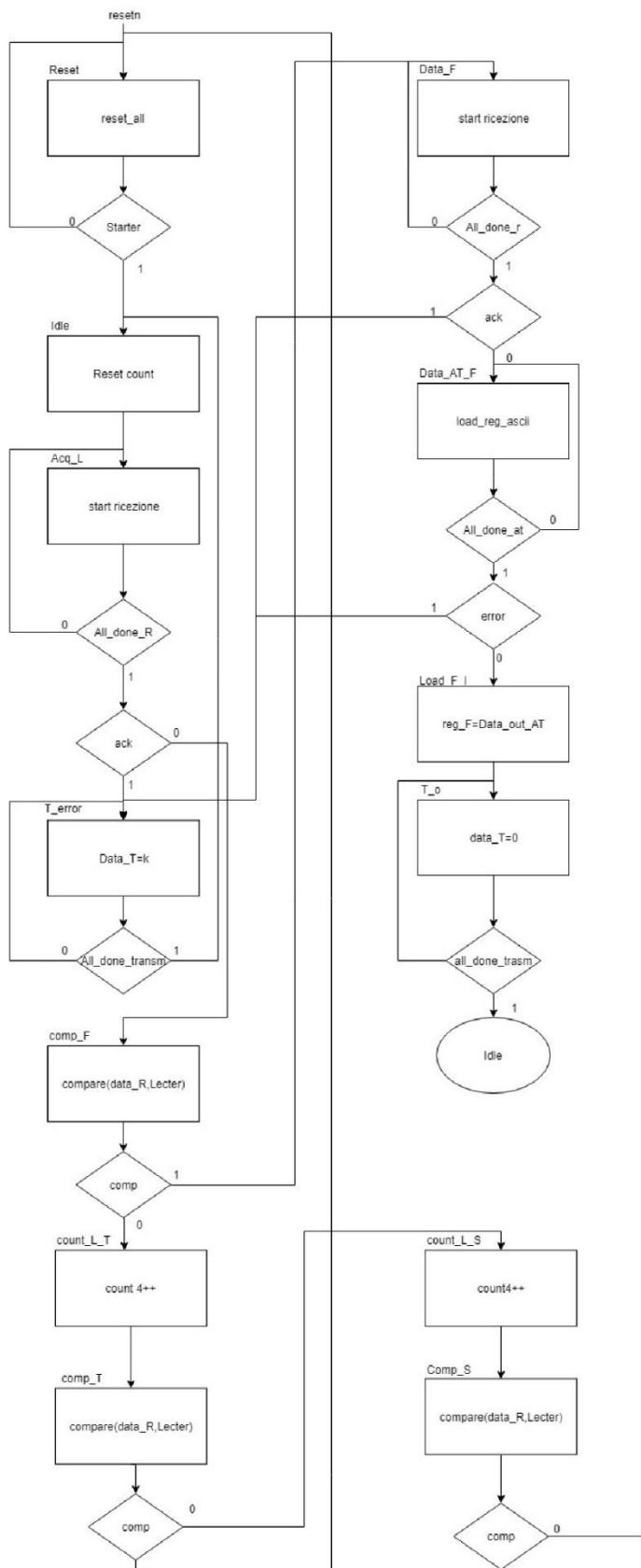
```

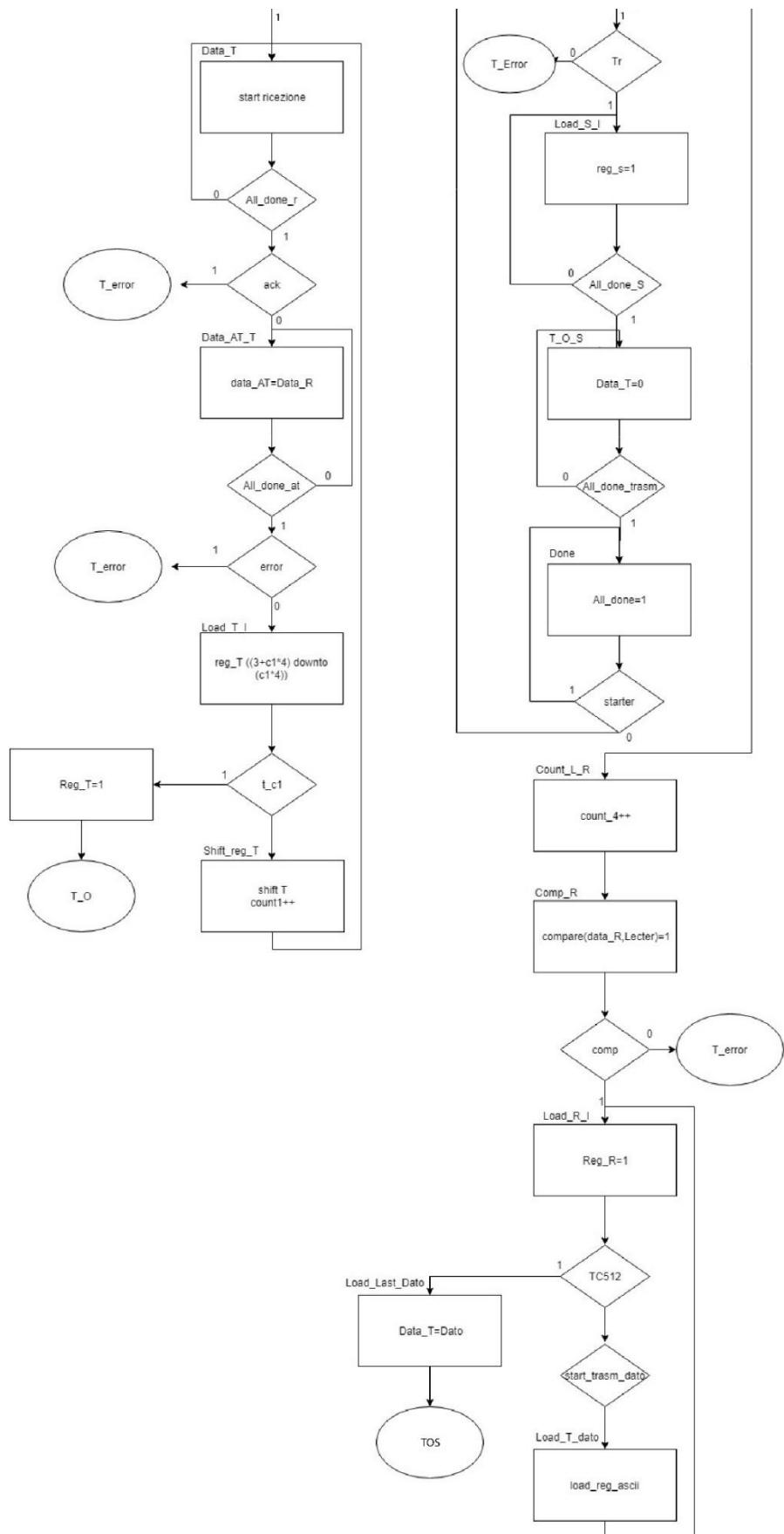
Datapath



Datapath della PC interface

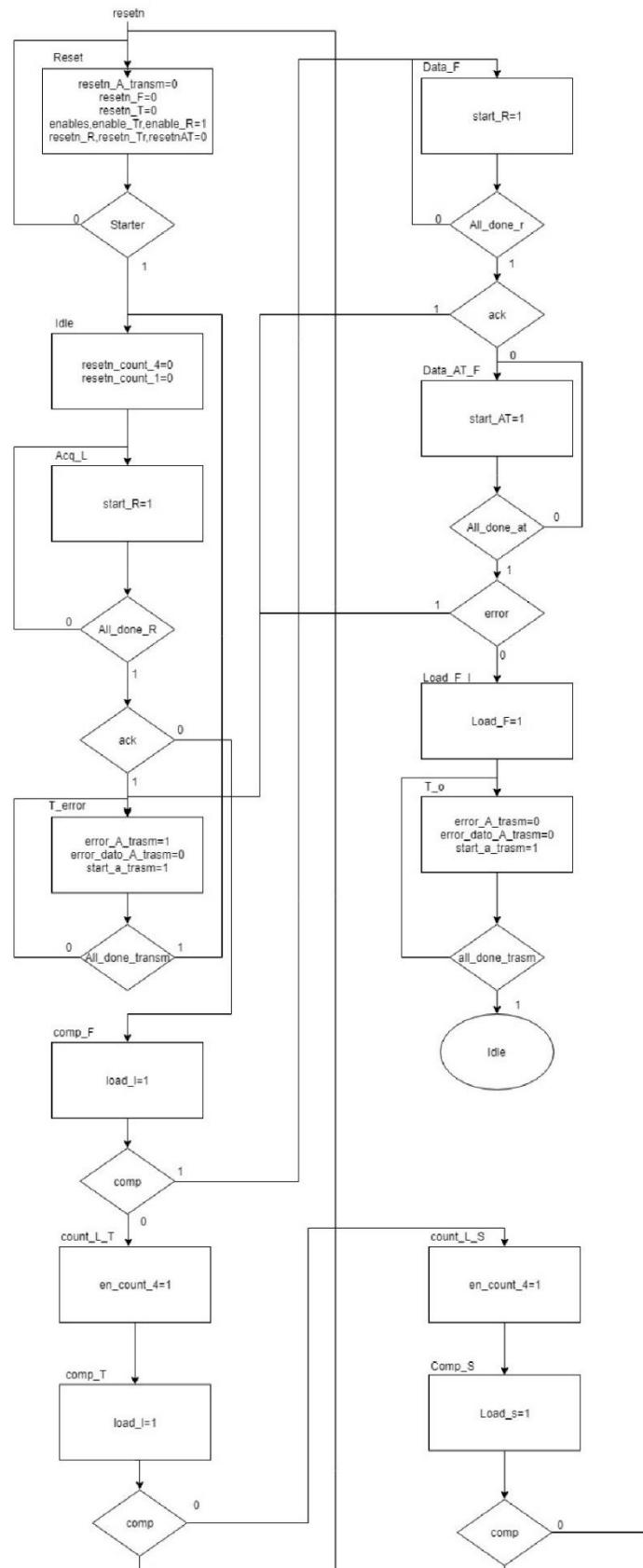
ASM chart

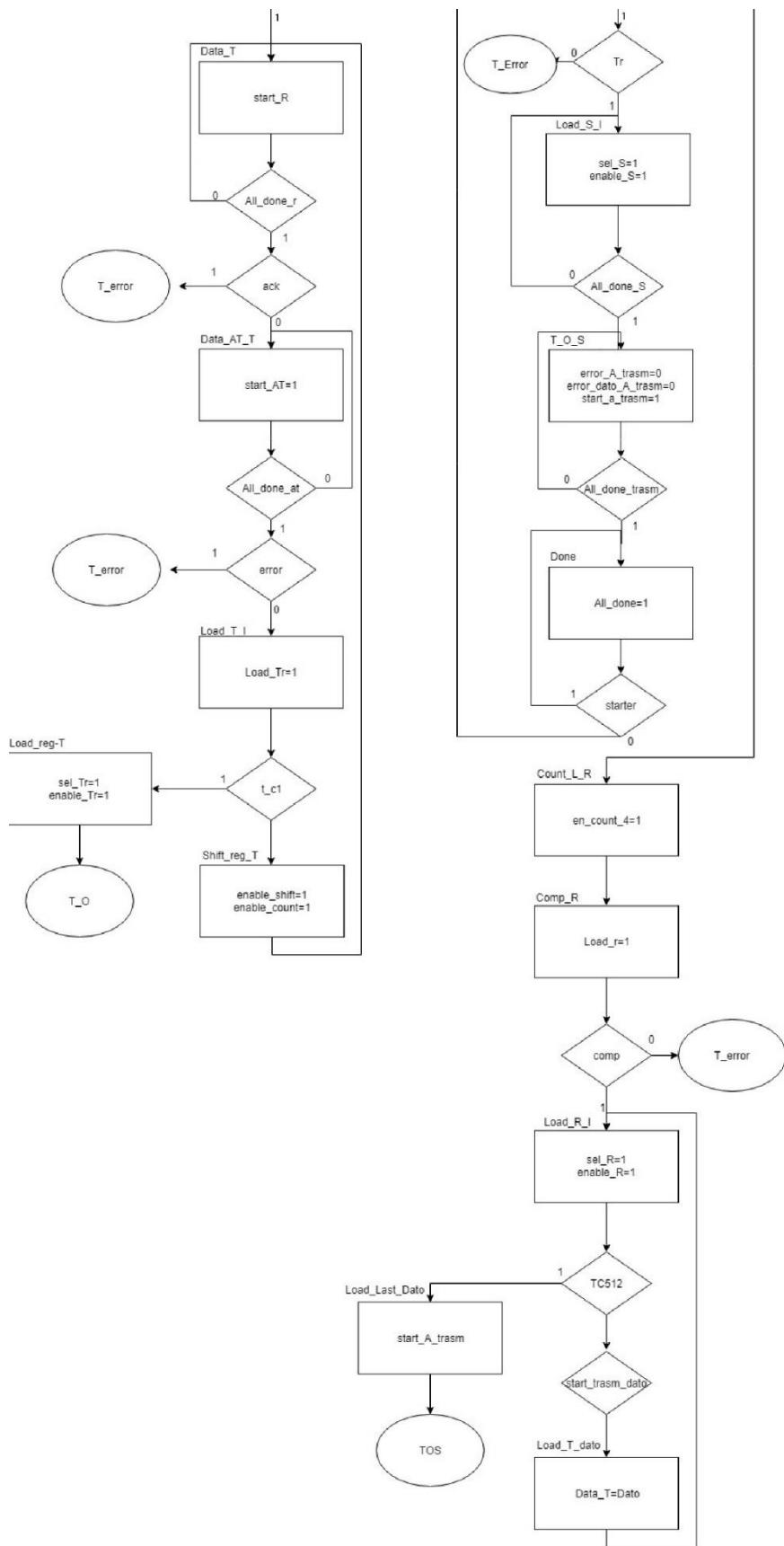




ASM chart della PC interface

Control ASM chart

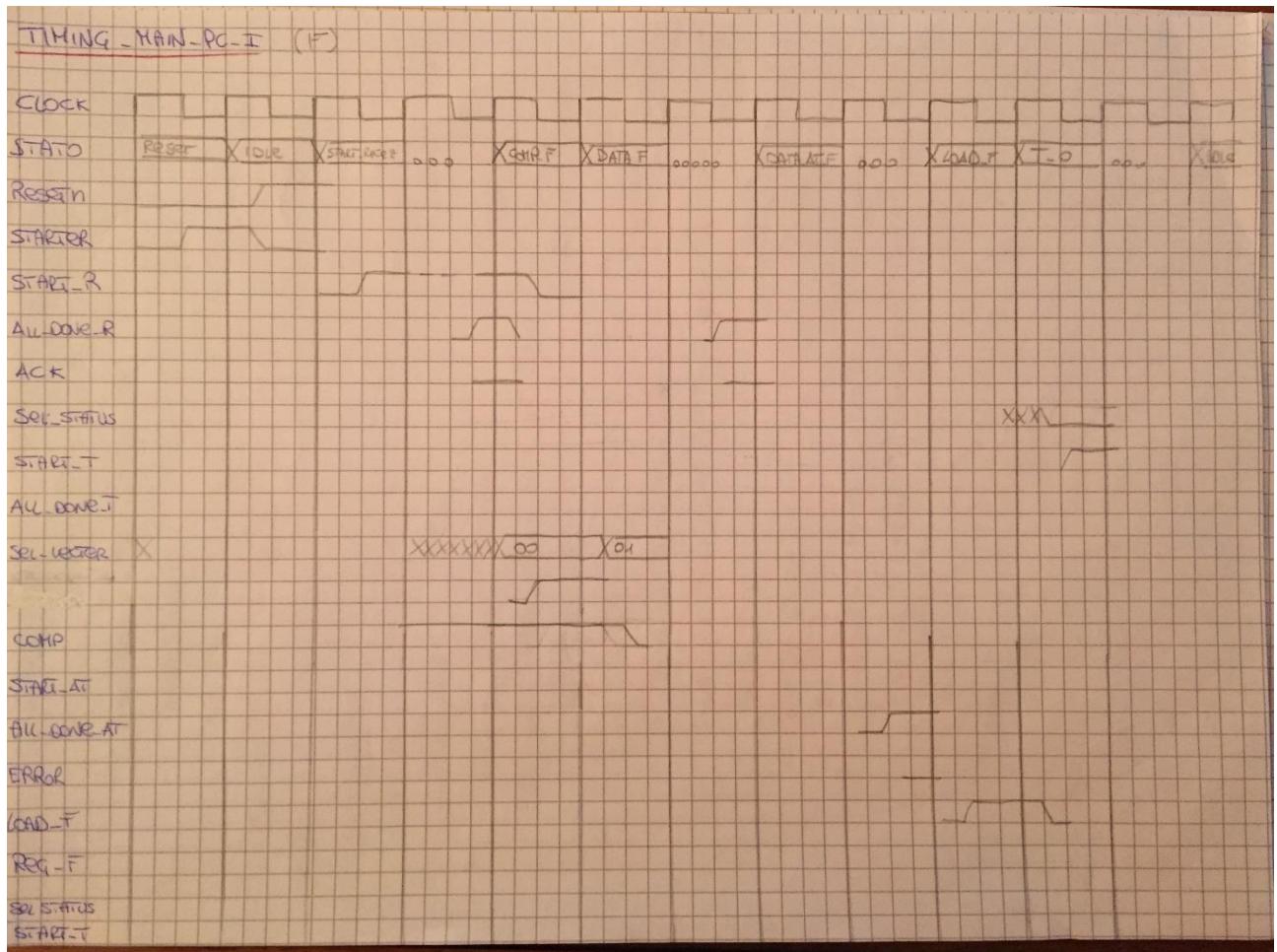




ASM chart dei controlli della PC interface

Timing

NOTA: Il timing è riferito all'acquisizione della lettera F e del valore numerico del prescaler.

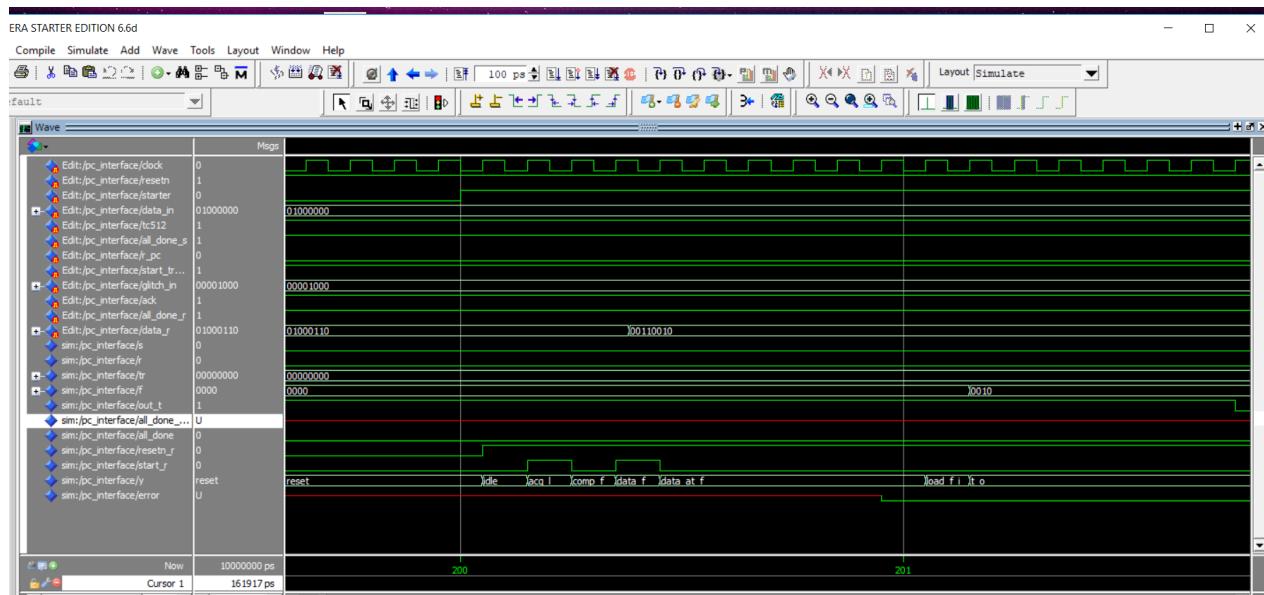


Timing della PC interface

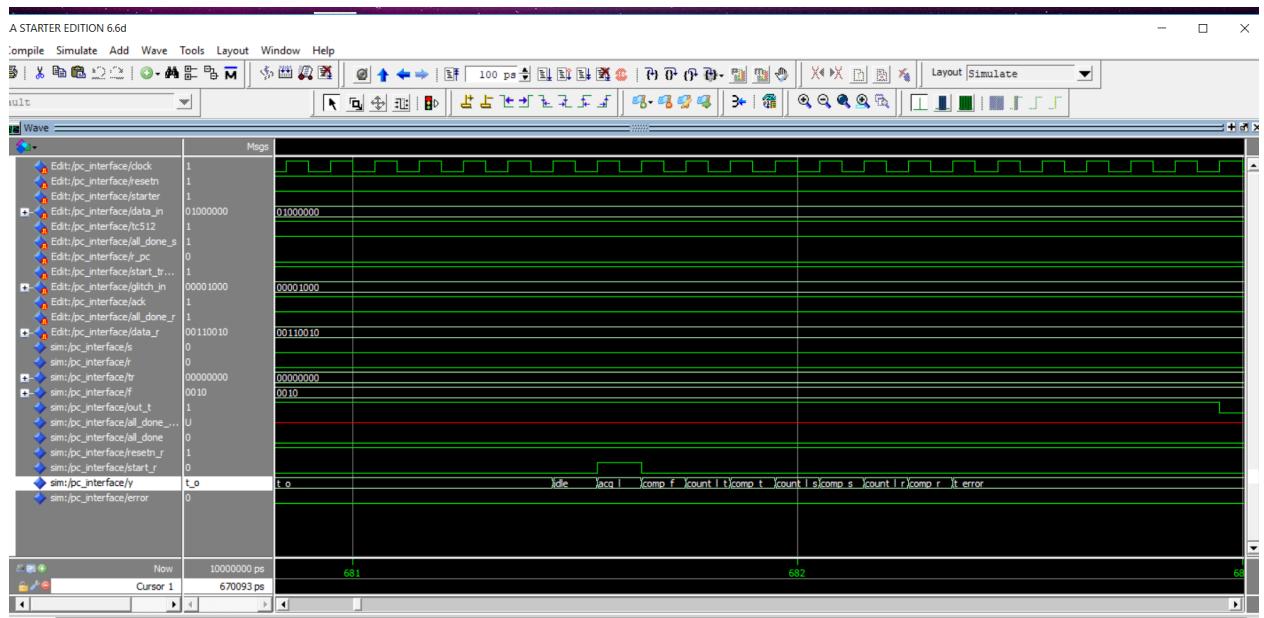
Simulazioni

NOTA: la simulazione è stata fatta nello stesso caso descritto nel timing.

Si è fatta prima una prova sul corretto rilevamento e successivamente una simulazione sul caso in cui la lettera non si valida.



Simulazione sul caso di acquisizione della lettera 'P' e del valore numerico del prescaler



Simulazione sul caso di acquisizione della lettera 'P' nel caso di parametro non valido del valore del prescaler

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

File VHDL : PC_interface.vhdl

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY Pc_interface IS
PORT(      Starter: IN STD_LOGIC;
Clock,Resetn: IN STD_LOGIC;
Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
TC512: IN STD_LOGIC;
All_done_s: IN STD_LOGIC;
R_Pc: IN STD_LOGIC;
Start_trasm_dato: IN STD_LOGIC;
glitch_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
S: OUT STD_LOGIC;
R: OUT STD_LOGIC;
Tr: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
F: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
out_T: OUT STD_LOGIC;
All_Done_A_Transmissione: OUT STD_LOGIC;
All_Done: OUT STD_LOGIC;
--test senza ricevitore
resetn_R: out std_logic;
start_R:out std_logic;
ack_R: in std_logic;
data_R: in std_logic_vector(7 downto 0);
all_done_r: in std_logic);
END Pc_interface;
--PORT( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
-- KEY: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
-- CLOCK50: IN STD_LOGIC;
-- UART_RXD: IN STD_LOGIC;
-- UART_TXD: OUT STD_LOGIC;
-- LEDR: OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
-- LEDG: OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
--END Pc_interface;
--
ARCHITECTURE behavior OF Pc_interface IS
--SIGNAL
starter,clock,resetn,TC512,All_done_s,R_pc,start_trasm_dato,s,r,out_T,all_done_A_transmissione,All_done: std_logic;
--signal Data_in,glitch_in,tr: std_logic_vector(7 downto 0);
--signal F: std_logic_vector(3 downto 0);
```

```

TYPE State_type IS
(Reset,Idle,Acq_L,T_error,Comp_F,Data_F,Data_AT_F,Load_F_I,T_o,Count_L_T,Comp_T,Data_T,Data_
AT_T,Load_T_I,Load_reg_T,Shift_reg_T,Count_L_S,Comp_S,Load_S_I,T_o_S,count_L_R,Comp_R,Load
_R_I,Load_last_dato,Load_T_dato,Done);
SIGNAL y : State_type;
SIGNAL
comp,en_count_4,resetn_count_4,resetn_T,Start_T,Resetn_A_T,resetn_A_transm,error_dato_A_transm,start
_A_T,error,All_done_T,All_done_A_T,resetn_Tr,load_Tr,enable_shift_Tr,en_count_1,resetn_count_1,tc_1,
resetn_F,Load_F,s_i,R_i,Tr_i,sel_s,sel_R,sel_Tr,enable_S,enable_Tr,enable_R,in_reg_S,in_reg_R,in_reg_Tr,r
esetn_reg_s,Resetn_reg_r,resetn_reg_Tr,All_done_A_transm,T_control,start_A_transm,error_A_Transm:
STD_LOGIC; --resetn_R,start_R,ack_R,all_done_R,
SIGNAL data_in_c,L,data_in_T: std_logic_vector(7 downto 0);--data_R,
SIGNAL data_out_A_T: std_logic_vector(3 downto 0);
SIGNAL Sel_Lecter: std_logic_vector(1 downto 0);
SIGNAL out_count_1: std_logic_vector(0 downto 0);

```

```

--COMPONENT
COMPONENT flipflop IS
PORT (
D, Clock , Resetn,Load : IN STD_LOGIC;
Q :OUT STD_LOGIC);
END COMPONENT;

```

```

COMPONENT mux2to1 IS
PORT (x, y, s : IN STD_LOGIC;
m : OUT STD_LOGIC);
END COMPONENT;

```

```
COMPONENT reg_asynch_reset_4bit is
```

```

port
(
    clk          : in std_logic;
    resetn      : in std_logic;
    pr_in       : in std_logic_vector(3 DOWNTO 0);
    load_parallel : in std_logic;
    pr_out      : out std_logic_vector(3 downto 0)
);

```

```
end component;
```

```

COMPONENT comparator_8bit IS
PORT (A,B: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
U: OUT STD_LOGIC);
END component;

```

```

COMPONENT counterupN IS
GENERIC (N : integer:=10);
PORT(   Enable : IN STD_LOGIC;

```

```

Clock , Resetn : IN STD_LOGIC;
      Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;

COMPONENT uart_ricevitore IS
PORT ( start_ricevitore_uart: IN STD_LOGIC;
clock,resetN_uart_ricevitore: IN STD_LOGIC;      --SONO DUE SEGNALI DI START E RESET SOLO
DEL RICEVITORE NON QUELLO GENERALE, MANDATO DALLA C.U generale
      r: IN STD_LOGIC;
      done : OUT STD_LOGIC;
      data_seriali : out std_logic_vector ( 7 downto 0);
      ack : OUT STD_LOGIC );
END COMPONENT;

COMPONENT ascii_translator IS
PORT(   Starter: IN STD_LOGIC;
      Clock,Resetn: IN STD_LOGIC;
      Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      Data_out: OUT STD_LOGIC_vector(3 downto 0);
      Error: OUT STD_LOGIC;
      All_Done: OUT STD_LOGIC);
END COMPONENT;

COMPONENT mux4to1_8bit IS
PORT (x, y, w,z : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      s: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
      m : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

COMPONENT shift_reg_4bit_asynch_reset is

port
(
      clk           : in std_logic;
      enable        : in std_logic;
      resetn       : in std_logic;
      pr_in         : in std_logic_vector(3 DOWNTO 0);
      load_parallel : in std_logic;
      pr_out        : out std_logic_vector(7 downto 0)
);
end COMPONENT;

```

```

COMPONENT Ascii_transmission IS
PORT(   Starter: IN STD_LOGIC;
      Clock,Resetn: IN STD_LOGIC;
      Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      Glitch_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      error: IN STD_LOGIC;

```

```

error_dato: IN STD_LOGIC;
all_done_T: IN STD_LOGIC;
Full_out: OUT STD_LOGIC;
Data_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
All_Done: OUT STD_LOGIC);
END COMPONENT;

```

```

COMPONENT trasmettitore IS
PORT( Starter: IN STD_LOGIC;
Clock,Resetn:IN STD_LOGIC;
Data: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
T: OUT STD_LOGIC;
All_Done: OUT STD_LOGIC);
END COMPONENT;

```

BEGIN

```

FSM_transitions:
PROCESS ( Clock , Resetn )
--VARIABLE first_state : std_logic :='0';

```

BEGIN

```

IF Resetn = '0' THEN
  y <= reset;
ELSIF (Clock 'EVENT AND Clock = '1') THEN
  CASE y IS
    WHEN Reset => IF starter = '1' THEN y <= Idle ;
      ELSE y <= Reset ;
      END IF ;
    WHEN Idle => y <= Acq_L ;
    WHEN Acq_L => IF all_done_R = '1' THEN IF Ack='1' THEN y <= T_error;
      ELSE y<=comp_F;
      END IF;
      ELSE y <= Acq_L ;
      END IF ;
    WHEN T_error => IF All_done_A_Transm = '1' THEN y <= Idle ;
      ELSE y <= T_error ;
      END IF ;
    WHEN Comp_F => IF Comp = '1' THEN y <= Data_F ;
      ELSE y <= Count_L_T;
      END IF ;
    WHEN Data_F=> IF all_done_R = '1' THEN IF Ack='1' THEN y <= T_error;
      ELSE y<=Data_AT_F;
      END IF;
      ELSE y <= Data_F ;
      END IF ;
    WHEN Data_AT_F=> IF all_done_A_T = '1' THEN IF error='1' THEN y <= T_error;
      ELSE y<=Load_F_I;
      END IF;
  END CASE;
END IF;

```

```

    ELSE y <= Data_AT_F ;
    END IF ;
WHEN Load_F_I => y <= T_o ;
WHEN T_o => IF All_done_A_Transm = '1' THEN y <= Idle ;
    ELSE y <= T_o ;
    END IF ;
WHEN Count_L_T => y <= Comp_T ;
WHEN Comp_T => IF Comp = '1' THEN y <= Data_T ;
    ELSE y <= Count_L_S;
    END IF ;
WHEN Data_T=> IF all_done_R = '1' THEN IF Ack='1' THEN y <= T_error;
    ELSE y<=Data_AT_T;
    END IF;
    ELSE y <= Data_T ;
    END IF ;
WHEN Data_AT_T=> IF all_done_A_T = '1' THEN IF error='1' THEN y <= T_error;
    ELSE y<=Load_T_I;
    END IF;
    ELSE y <= Data_AT_T ;
    END IF ;
WHEN Load_T_I => IF TC_1 = '1' THEN y <= Load_reg_T ;
    ELSE y <= Shift_reg_T;
    END IF ;
WHEN Load_reg_T => y <= T_o ;
WHEN Shift_reg_T=>y <= Data_T ;
WHEN Count_L_S => y <= Comp_S ;
WHEN Comp_S=> IF Comp <= '1' THEN IF T_control<='1' THEN y <= Load_S_I;
    ELSE y<=T_error;
    END IF;
    ELSE y <= Count_L_R ;
    END IF ;
WHEN Load_S_I=>IF All_done_S = '1' THEN y <= T_o_S ;
    ELSE y <= Load_S_I;
    END IF ;
WHEN T_o_S=> IF All_Done_A_Transm= '1' THEN y <= Done ;
    ELSE y <= T_o_S;
    END IF ;
WHEN Count_L_R => y <= Comp_R ;
WHEN Comp_R => IF Comp = '1' THEN y <= Load_R_I ;
    ELSE y <= T_error;
    END IF ;
WHEN Load_R_I => IF TC512 = '1' THEN y <= Load_last_dato ;
    ELSE IF Start_trasm_dato='1' then y<=Load_T_Dato;
        ELSE y<=Load_R_I;
        END IF;
    END IF ;
WHEN Load_last_dato => y <= T_o_S ;
WHEN Load_T_dato =>IF All_done_A_transm = '1' THEN y <= Load_R_I ;
    ELSE y <= Load_T_dato;

```

```

        END IF ;
WHEN Done => IF Starter= '1' THEN y <= Done ;
ELSE y <= Reset;
END IF ;
END CASE;
END IF;
END PROCESS;

```

FSM_outputs: PROCESS(y)

```

BEGIN
resetn_F<='1';
Load_F<='0';
resetn_Tr<='1';
resetn_Tr<='0';
en_count_1<='0';
resetn_count_1<='1';
enable_s<='0';
enable_tr<='0';
enable_r<='0';
resetn_R<='1';
start_R<='0';
resetn_T<='1';
resetn_A_T<='1';
start_A_T<='0';
En_count_4<='0';
resetn_count_4<='1';
All_Done<='0';
start_A_Transm<='0';
error_A_transm<='0';
Resetn_A_transm<='1';
Error_dato_a_transm<='0';
Resetn_reg_s<='1';
Resetn_reg_r<='1';
Resetn_reg_tr<='1';

```

CASE y IS

WHEN Reset=> Resetn_F <= '0';

```

    Resetn_T <= '0';
    Resetn_R <= '0';
    Resetn_Tr <= '0';
    Resetn_A_T <= '0';
    enable_R<='0';
    enable_S<='0';
    enable_Tr<='0';
    Resetn_A_Transm<='0';
    Resetn_reg_s<='0';
    Resetn_reg_r<='0';
    Resetn_reg_tr<='0';

```

```

WHEN Idle => Resetn_count_4 <= '0';
    Resetn_count_1 <= '0';

WHEN Acq_L => Start_R <= '1';

WHEN T_error => Error_A_transm <= '1';
    Error_dato_A_transm<='1';
    Start_A_Transm<='1';

WHEN Comp_F=>En_count_4<='0';

WHEN Comp_T=>En_count_4<='0';

WHEN Comp_S=>En_count_4<='0';

WHEN Comp_R=>En_count_4<='0';

WHEN Data_F => Start_R <= '1';

WHEN Data_AT_F => Start_A_T <= '1';

WHEN Load_F_I => Load_F <= '1';

WHEN T_o => Error_A_transm <= '0';
    Error_dato_A_transm<='0';
    Start_A_Transm<='1';

WHEN Count_L_T=>En_Count_4<='1';

WHEN Data_T => Start_R <= '1';

WHEN Data_AT_T => Start_A_T <= '1';

WHEN Load_T_I => Load_Tr <= '1';

WHEN Load_Reg_T => Sel_Tr <= '1';
    enable_Tr<='1';

WHEN Shift_Reg_T => Enable_shift_tr <= '1';
    En_Count_1<='1';

WHEN Count_L_S=> En_Count_4<='1';

WHEN Load_S_I => Sel_S <= '1';
    enable_S<='1';

WHEN T_o_S=>Error_A_transm <= '0';

```

```
Error_dato_A_transm<='0';
Start_A_Transm<='1';
```

```
WHEN Count_L_R=> En_Count_4<='1';
```

```
WHEN Load_R_I => Sel_R <= '1';
enable_R<='1';
```

```
WHEN Load_last_Dato => Start_A_transm <= '1';
```

```
WHEN Load_T_Dato => Start_A_transm <= '1';
```

```
WHEN Done => All_Done <= '1';
```

```
END CASE;
END PROCESS;
```

```
Count_4: counterupN GENERIC MAP(2)
PORT MAP(En_count_4,Clock,Resetn_count_4,Sel_Lecter);
```

```
Count_1: counterupN GENERIC MAP(1)
PORT MAP(En_count_1,Clock,Resetn_count_1,out_count_1);
```

```
Mux_L: mux4to1_8bit PORT MAP
("01000110","01010100","01010011","01010010",sel_lester,L);
```

```
Comparator: comparator_8bit PORT MAP
(Data_R,L,Comp);
```

```
Reg_F: reg_asynch_reset_4bit PORT MAP
(Clock,Resetn_F,Data_out_A_T,Load_F,F);
```

```
Reg_Tr: shift_reg_4bit_asynch_reset PORT MAP
(Clock,Load_Tr,Resetn_Tr, Data_out_A_T,enable_shift_Tr,Tr);
```

```
--Rx: uart_ricevitore PORT MAP
-- (Start_R,Clock,Resetn_R,R_Pc,All_done_R,Data_R,NOT Ack_R);
```

```
Tx: trasmettitore PORT MAP
( start_T,Clock,Resetn_T,Data_in_T,Out_T,All_done_T);
```

```
Ascii_trasm: Ascii_transmission PORT MAP
(
Start_A_transm,Clock,Resetn_A_transm,Data_in,Glitch_in,error_A_Transm,error_Dato_A_transm,All_done_T,Start_T,Data_in_T,All_Done_A_transm);
```

```
Ascii_transl: ascii_translator PORT MAP
(Start_A_T,Clock,Resetn_A_T,Data_R,Data_out_A_T, error,All_Done_A_T);
```

```

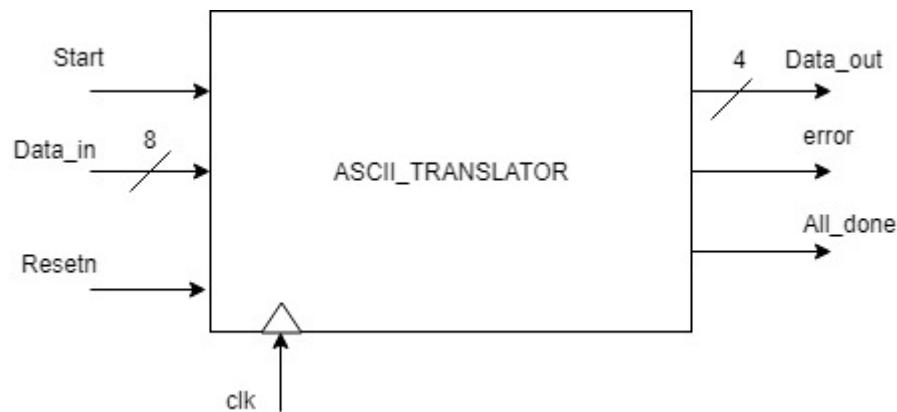
mux_s: mux2to1 PORT MAP
  ('0','1',sel_s,in_reg_s);
mux_r: mux2to1 PORT MAP
  ('0','1',sel_r,in_reg_r);
mux_Tr: mux2to1 PORT MAP
  ('0','1',sel_Tr,in_reg_Tr);

Reg_s: flipflop PORT MAP
  (in_reg_s,Clock,Resetn_reg_s,enable_s,S);
Reg_r: flipflop PORT MAP
  (in_reg_r,Clock,Resetn_reg_r,enable_r,R);
Reg_Tr_C: flipflop PORT MAP
  (in_reg_Tr,Clock,Resetn_reg_Tr,enable_Tr,T_control);

-- starter<=KEY(0);
-- clock<=SW(17);
-- resetn<=KEY(1);
-- data_in<=SW(7 downto 0);
--GLITCH_IN<=SW(15 DOWNTO 8);
--TC512<=KEY(2);
--ALL_DONE_S<=KEY(3);
--START_TRASM_DATO<=SW(16);
--R_PC<=UART_RXD;
--LEDR(11 downto 4)<=TR(7 downto 0);
-- LEDR(3 downto 0)<=F(3 downto 0);
--LEDG(0)<=S;
--LEDG(1)<=R;
--UART_TXD<=OUT_T;
--LEDG(3)<=ALL_DONE_A_TRANSMISSIONE;
--LEDG(4)<=ALL_DONE;
END behavior;

```

ASCII translator



Blocco dell'ASCII translator

L'Ascii Translator ha 2 compiti:

- riconoscere se è stato inviato un dato non valido
- Tradurre il dato da codice ascii a binario

Il dato in ingresso è inserito in un registro a 8 bit.

Primo step di controllo errore: I bit 7 downto 4 devono essere “0011” in caso sia stato ricevuto un codice Ascii di una cifra(0-9). Devono invece essere “0100” in caso sia stata ricevuta una lettera(A-F). Due comparatori verificano questa uguaglianza e segnalano l’eventuale errore(err).

Secondo step di controllo errore: Un circuito totalmente combinatorio costituito da porte logiche elementari verifica che i bit 3 downto 0 rappresentino effettivamente una cifra da 0 a 9 oppure una lettera da A a F. in caso contrario viene segnalato l’errore.

Traduzione del dato da codice Ascii a codice binario: Per la traduzione vengono presi gli ultimi 4 bit del dato in ingresso(i bit meno significativi).

In caso il dato in ingresso sia una lettera(riconosciuta dai comparatori) ai 4 bit viene sommato 9 in modo da tradurre correttamente il dato (es.A=10-> ascii=0001 binario=1010).

In caso il dato in ingresso sia una numero(riconosciuto dai comparatori) ai 4 bit viene sommato 0.

In uscita è fornito il dato tradotto e un segnale di errore.

Pseudocode

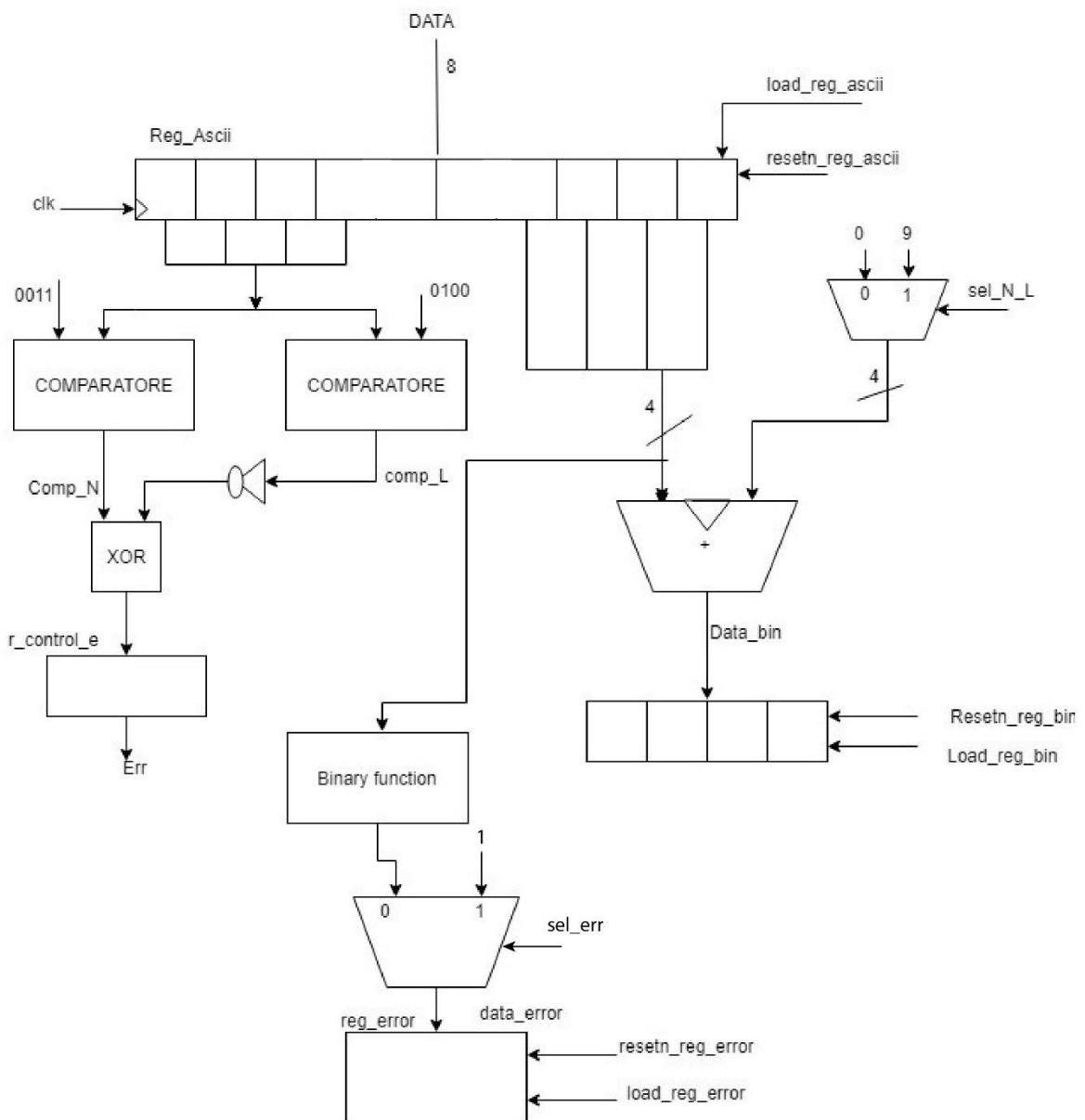
```
Reg_ascii=0;  
Reg_error=0;  
If start=1  
    Reg_ascii=data_in;  
    If reg_ascii(7 downto 4)=0011  
        Reg_bin=reg_ascii(3 downto 0)+0;  
        Reg_error=Reg_ascii(3) AND(Reg_ascii(2) OR Reg_ascii(1));  
    Else if (Reg_ascii(7 downto 4)=0100  
        Reg_bin=Reg_ascii(3 downto 0)+9;  
        Reg_error=Reg_ascii(3) OR( Reg_ascii(2) AND Reg_ascii(1) AND      Reg_ascii(0);  
    Else  
        Reg_error=1;
```

Datapath

Binary function: un mux con sel_N_L come selezionatore seleziona:

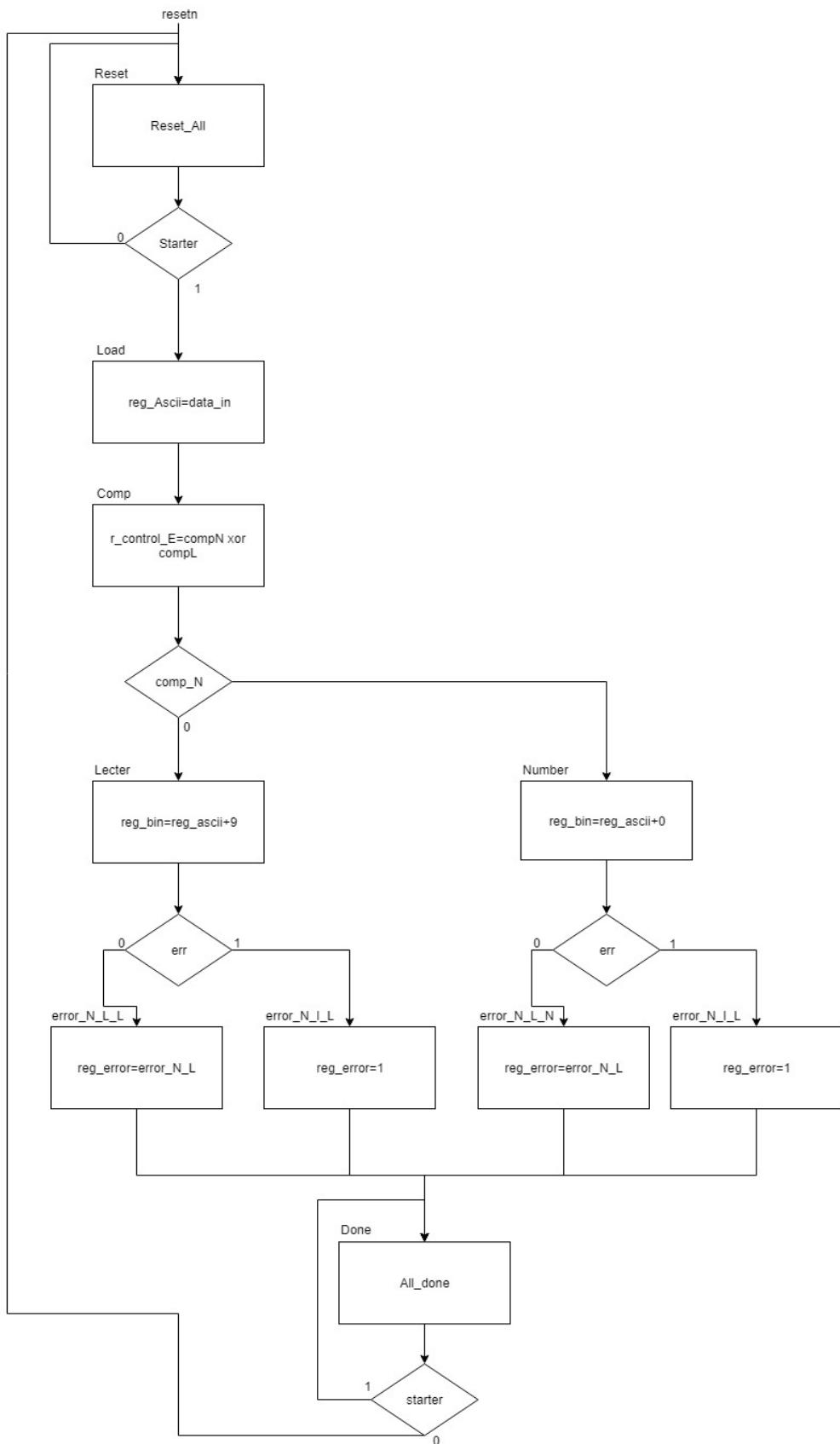
If sel_N_L=0: Reg_error=Reg_ascii(3) OR(Reg_ascii(2) AND Reg_ascii(1) AND Reg_ascii(0));

Il sel_N_L=1: Reg_error=Reg_ascii(3) AND(Reg_ascii(2) OR Reg_ascii(1))



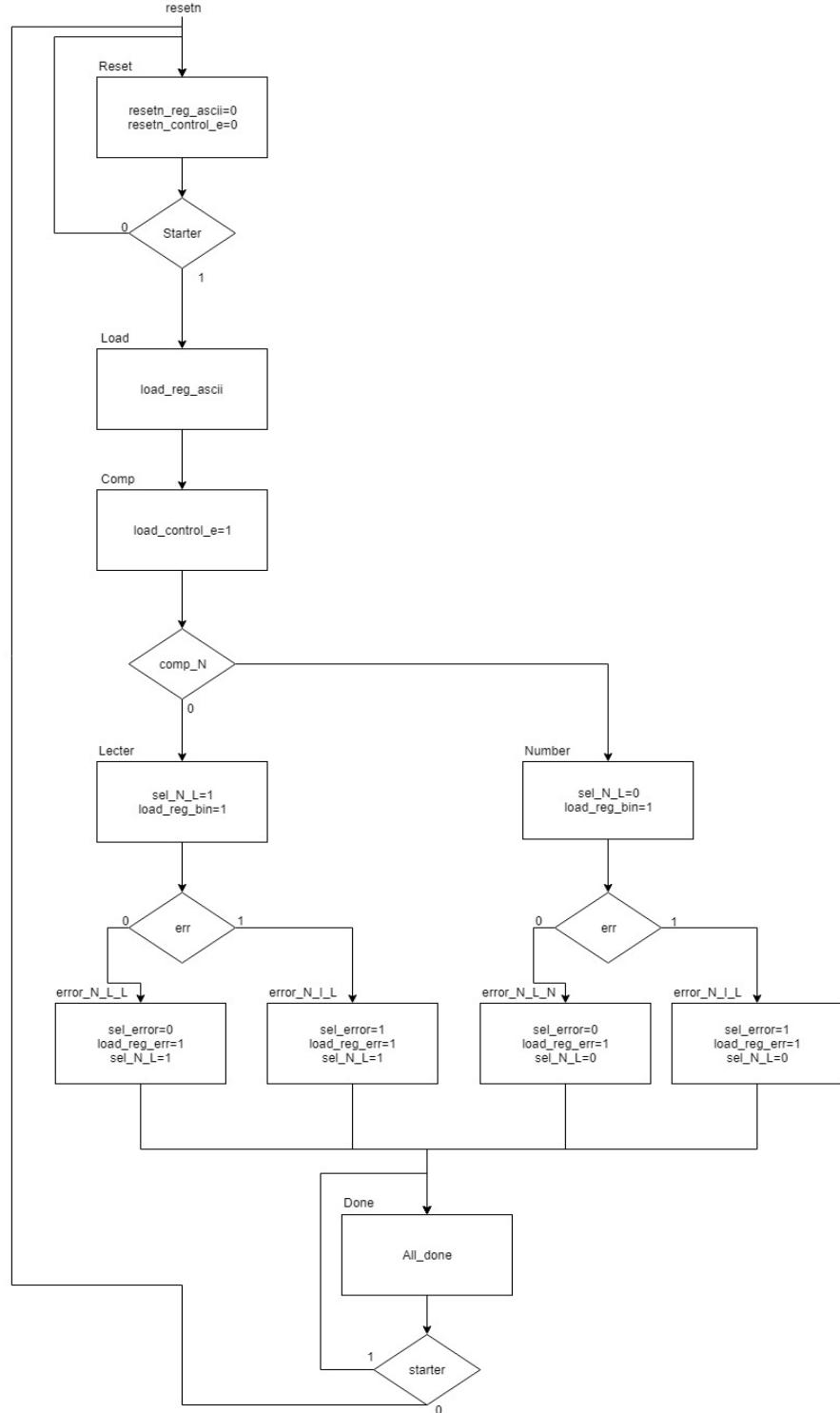
Datapath dell'ASCII translator

ASM chart



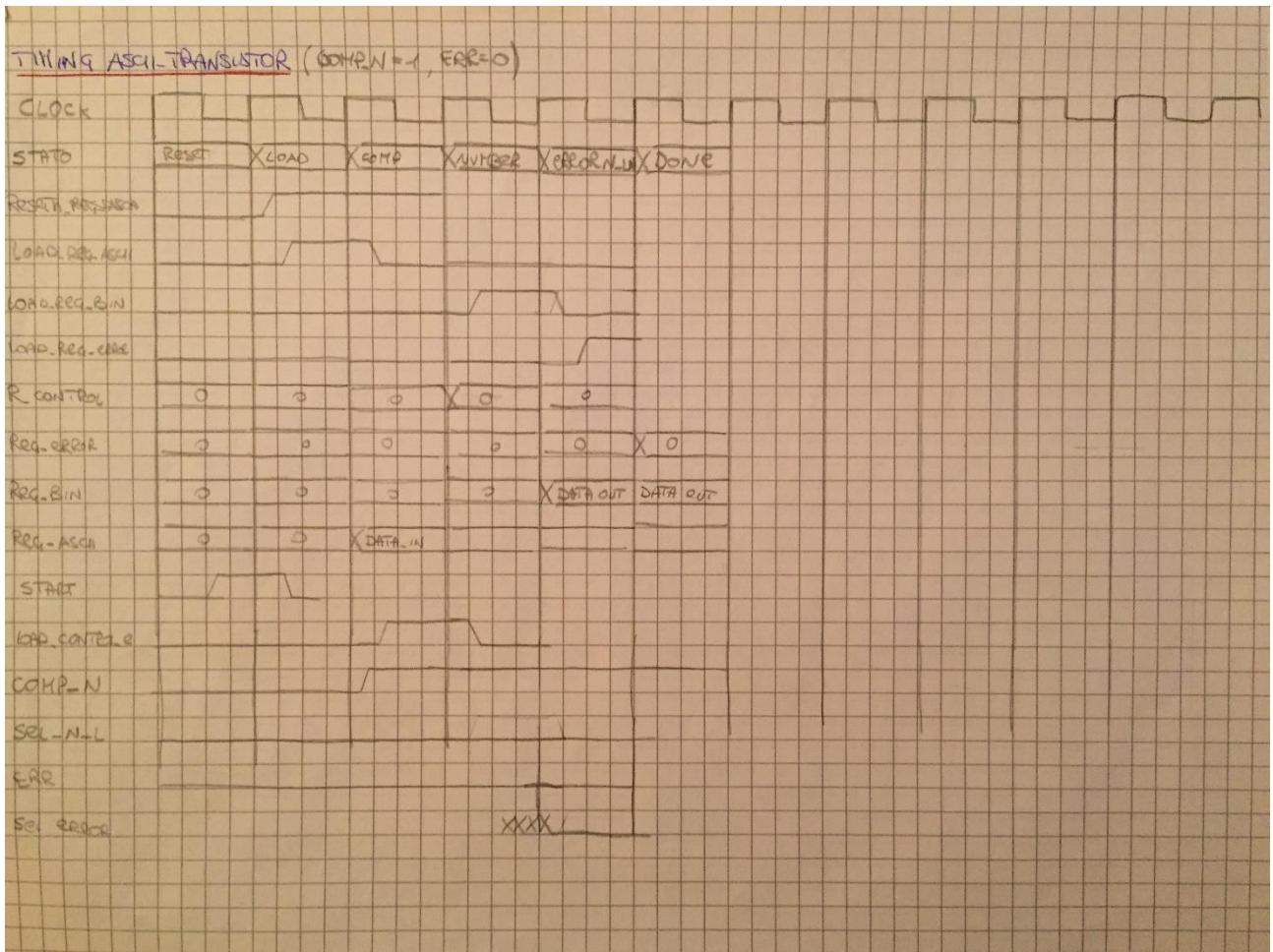
ASM chart dell'ASCII translator

Control ASM chart



ASM chart dei controlli dell'ASCII translator

Timing



Timing dell'ASCII translator

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore interno a QUARTUS II.

File VHDL: ASCII_translator.vhd

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY ascii_translator IS
PORT( Starter: IN STD_LOGIC;
      Clock,Resetn: IN STD_LOGIC;
      Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      Data_out: OUT STD_LOGIC_vector(3 downto 0);
      Error: OUT STD_LOGIC;
      All_Done: OUT STD_LOGIC);

```

```

END ascii_translator;
--PORT( SW: IN STD_LOGIC_VECTOR(8 DOWNTO 0);
--      KEY:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
--      LEDR: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
--      LEDG: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
--END ascii_translator;

```

ARCHITECTURE behavior OF ascii_translator IS

--SIGNAL

--SIGNAL starter,clock,resetn,All_done,Error: std_logic;

--signal Data_in: std_logic_vector(7 downto 0);

--signal Data_out: std_logic_vector(3 downto 0);

TYPE State_type IS

(Reset,Load,Comp,Lecter,Number,Error_N_L_N,Error_1_N,Error_N_L_L,Error_1_L,Done);

SIGNAL y : State_type;

SIGNAL load_reg_ascii,

resetn_reg_ascii,resetn_reg_bin,load_reg_bin,resetn_reg_error,load_reg_error,sel_error,sel_N_L,resetn_r_control_e,load_r_control_e,comp_N,comp_L,in_r_control_e,data_error,err: STD_LOGIC;

SIGNAL ascii_out: std_logic_vector(7 downto 0);

SIGNAL in_adder,in_reg_bin: std_logic_vector(3 downto 0);

--COMPONENT

COMPONENT adder4 IS

PORT (Carry_in : IN STD_LOGIC;

Input1 , Input2 : IN STD_LOGIC_VECTOR (3 DOWNTO 0);

Sum : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));

END COMPONENT;

COMPONENT flipflop IS

PORT (

D, Clock , Resetn,Load : IN STD_LOGIC;

Q :OUT STD_LOGIC);

END COMPONENT;

COMPONENT mux2to1 IS

PORT (x, y, s : IN STD_LOGIC;

m : OUT STD_LOGIC);

END COMPONENT;

COMPONENT reg_asynch_reset_8bit is

```

port
(
    clk          : in std_logic;
    resetn       : in std_logic;
    pr_in        : in std_logic_vector(7 DOWNTO 0);
    load_parallel : in std_logic;
    pr_out       : out std_logic_vector(7 downto 0)
);

```

```

end COMPONENT;

COMPONENT mux2to1_4bit IS
PORT (x, y: std_logic_vector(3 downto 0);
      s : IN STD_LOGIC;
      m : OUT STD_LOGIC_VECTOR(3 downto 0));
END COMPONENT;

COMPONENT reg_asynch_reset_4bit is

port
(
    clk          : in std_logic;
    resetn       : in std_logic;
    pr_in        : in std_logic_vector(3 DOWNTO 0);
    load_parallel : in std_logic;
    pr_out       : out std_logic_vector(3 downto 0)
);

end component;

COMPONENT comparator_4bit IS
PORT (A,B: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      U: OUT STD_LOGIC);
END component;

COMPONENT error_generator IS
PORT (data : IN STD_LOGIC_VECTOR(3 downto 0);
      sel_mux0,sel_mux1: IN STD_LOGIC;
      out_error: OUT STD_LOGIC);

END COMPONENT;

BEGIN

FSM_transitions:
PROCESS ( Clock , Resetn )
-VARIABLE first_state : std_logic :='0';

BEGIN

IF Resetn = '0' THEN
    y <= reset;
ELSIF (Clock 'EVENT AND Clock = '1') THEN
    CASE y IS
        WHEN reset => IF starter = '1' THEN y <= Load ;
                    ELSE y <= reset ;
                    END IF ;
        WHEN Load => y <= Comp ;
    END CASE;
END IF;
END PROCESS;
END;

```

```

WHEN Comp => IF comp_n = '1' THEN y <= Number ;
    ELSE y <= Lecter ;
    END IF ;
WHEN Number => IF err = '1' THEN y <= Error_1_N ;
    ELSE y <= Error_N_L_N ;
    END IF ;
WHEN Lecter => IF err = '1' THEN y <= Error_1_L ;
    ELSE y <= Error_N_L_L;
    END IF ;
WHEN Error_N_L_N => y <= Done ;
WHEN Error_N_L_L => y <= Done ;
WHEN Error_1_N => y <= Done ;
WHEN Error_1_L => y <= Done ;
WHEN Done => IF starter = '1' THEN y <= Done ;
    ELSE y <= reset ;
    END IF ;
END CASE;
END IF;
END PROCESS;

```

FSM_outputs: PROCESS(y)

BEGIN

```

resetn_reg_ascii<='1';
resetn_reg_bin<='1';
resetn_reg_error<='1';
resetn_r_control_e<='1';
sel_N_L<='0';
Load_reg_ascii<='0';
Load_r_control_e<='0';
Load_reg_bin<='0';
Load_reg_error<='0';
sel_error<='0';
All_Done<='0';

```

CASE y IS

```

WHEN reset=> Resetn_reg_ascii <= '0';
    Resetn_r_control_e<='0';
WHEN load => Load_reg_ascii <= '1';
WHEN comp => Load_r_control_e <= '1';
WHEN lecturer => Load_reg_bin <= '1';
    sel_N_L<='1';
WHEN number=> Load_reg_bin <= '1';
    sel_N_L<='0';
WHEN error_N_L_N => Load_reg_error <= '1';
    sel_error<='0';
    sel_N_L<='0';
WHEN error_N_L_L => Load_reg_error <= '1';
    sel_error<='0';
    sel_N_L<='1';

```

```

WHEN error_1_N => Load_reg_error <= '1';
    sel_error<='1';
    sel_N_L<='0';
WHEN error_1_L => Load_reg_error <= '1';
    sel_error<='1';
    sel_N_L<='1';
WHEN Done => All_Done <= '1';
END CASE;
END PROCESS;

```

```

reg_ascii: reg_asynch_reset_8bit
    PORT MAP(clock,resetn_reg_ascii,data_in,load_reg_ascii,ascii_out);

```

```

comparatore_1:comparator_4bit PORT MAP
    ("0011",ascii_out(7 downto 4),comp_N);
comparatore_2:comparator_4bit PORT MAP
    ("0100",ascii_out(7 downto 4),comp_L);
in_r_control_e<=comp_N XOR NOT Comp_L;

```

```

R_control_e: flipflop PORT MAP
    (in_r_control_e,clock,resetn_r_control_e,load_r_control_e,err);

```

```

Mux0: mux2to1_4bit
    PORT MAP("0000","1001", sel_N_L,in_adder);

```

```

adder: adder4 PORT MAP
    ('0',ascii_out(3 downto 0),in_adder(3 downto 0),in_reg_bin);

```

```

reg_bin: reg_asynch_reset_4bit PORT MAP
    (clock,resetn_reg_bin,in_reg_bin,load_reg_bin,Data_out);

```

```

error_gen: error_generator PORT MAP
    (ascii_out(3 downto 0),sel_N_L,sel_error,data_error);

```

```

reg_error: flipflop PORT MAP
    (data_error,clock,resetn_reg_error,load_reg_error>Error);

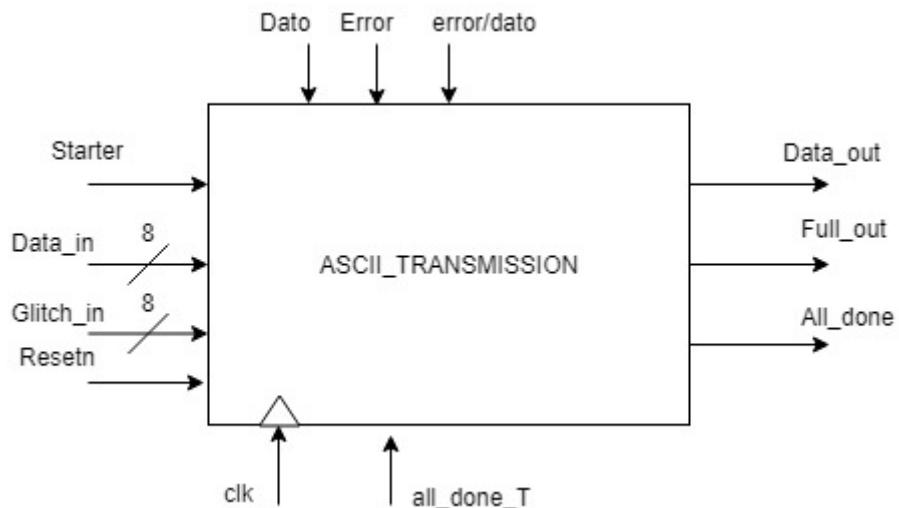
```

```

-- starter<=SW(8);
-- clock<=KEY(1);
-- resetn<=KEY(0);
-- data_in<=SW(7 downto 0);
-- LEDR(0)<=error;
-- LEDG(3 downto 0)<=Data_out(3 downto 0);
-- LEDR(1)<=All_done;
END behavior;

```

ASCII transmission



Blocco dell'ASCII trasmission

Blocco realizzato per tradurre il dato prima di inviarlo tramite il trasmettitore. Traduce da binario a codice Ascii.

Segnali di errore: In caso in ingresso vengano inviati dei segnali d'errore il uscita può essere inviato K. Se invece non c'è stato errore può essere inviato O.

Segnali di dato: Il dato è inserito in un registro a 8 bit, così come i glitch. Si procede bit a bit. In caso il glitch in posizione di invio sia 0 allora se il bit in posizione di invio di dato è '1' viene inviato il codice ascii per '1' altrimenti viene inviato '0'. Se il glitch in posizione di invio è invece '1' allora viene inviata X.

Dopo aver inviato tutti e 8 i simboli di dato viene inviato il codice ascii per new line in modo da ottenere i dati su schermo uno sotto l'altro.

Pseudocodice

Reg_number=0;

Reg_glitch=0;

While(starter=1)

If error_dato=0

If error=0

 Data_out=0;

Else

 Data_out=K;

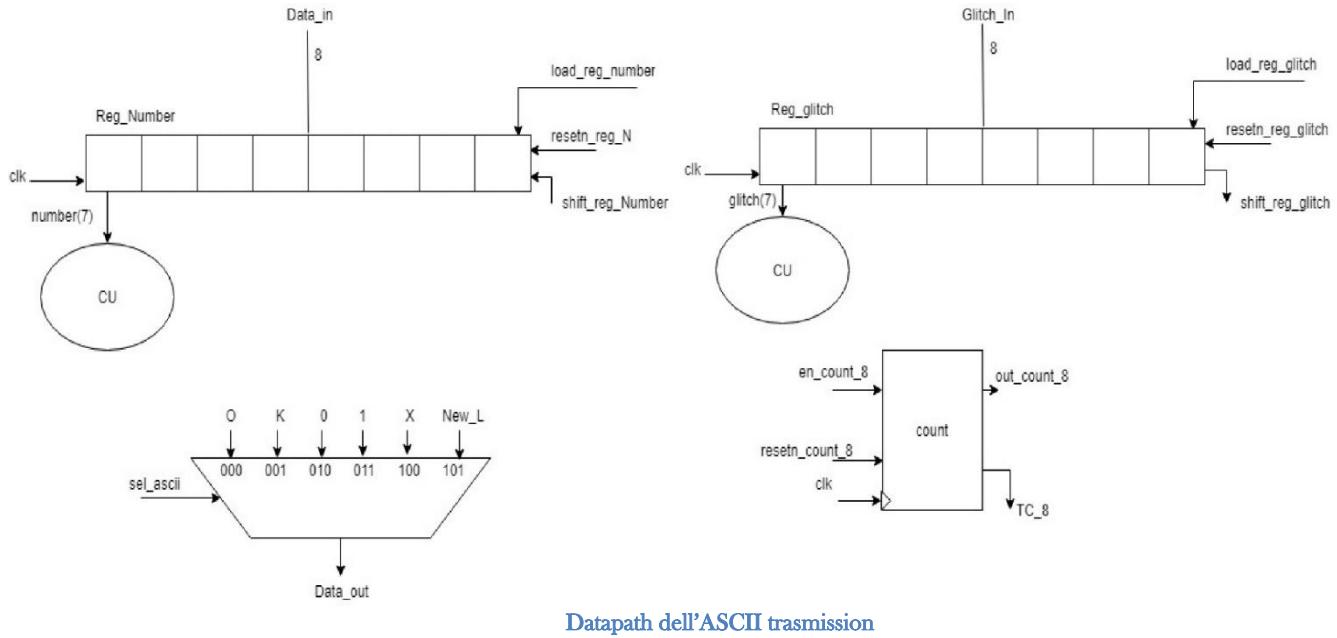
End if;

```

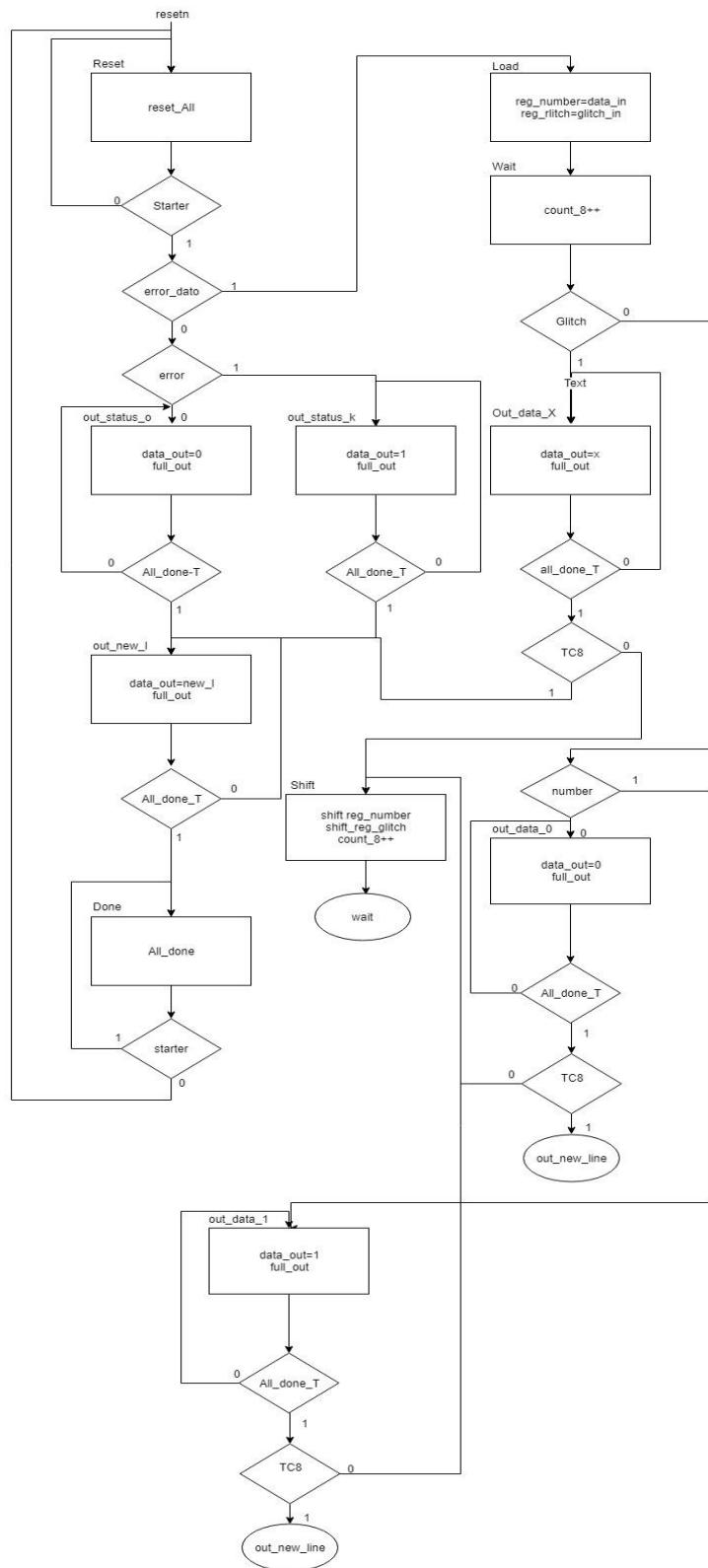
Else
For i= 7:0
    If glitch=1
        Data_out=X
    Else
        If number=0
            Data_out=0;
        Else
            Data_out=1;
        End if;
    End if;
    Data_out="New_line";
All_done=1;

```

Datapath

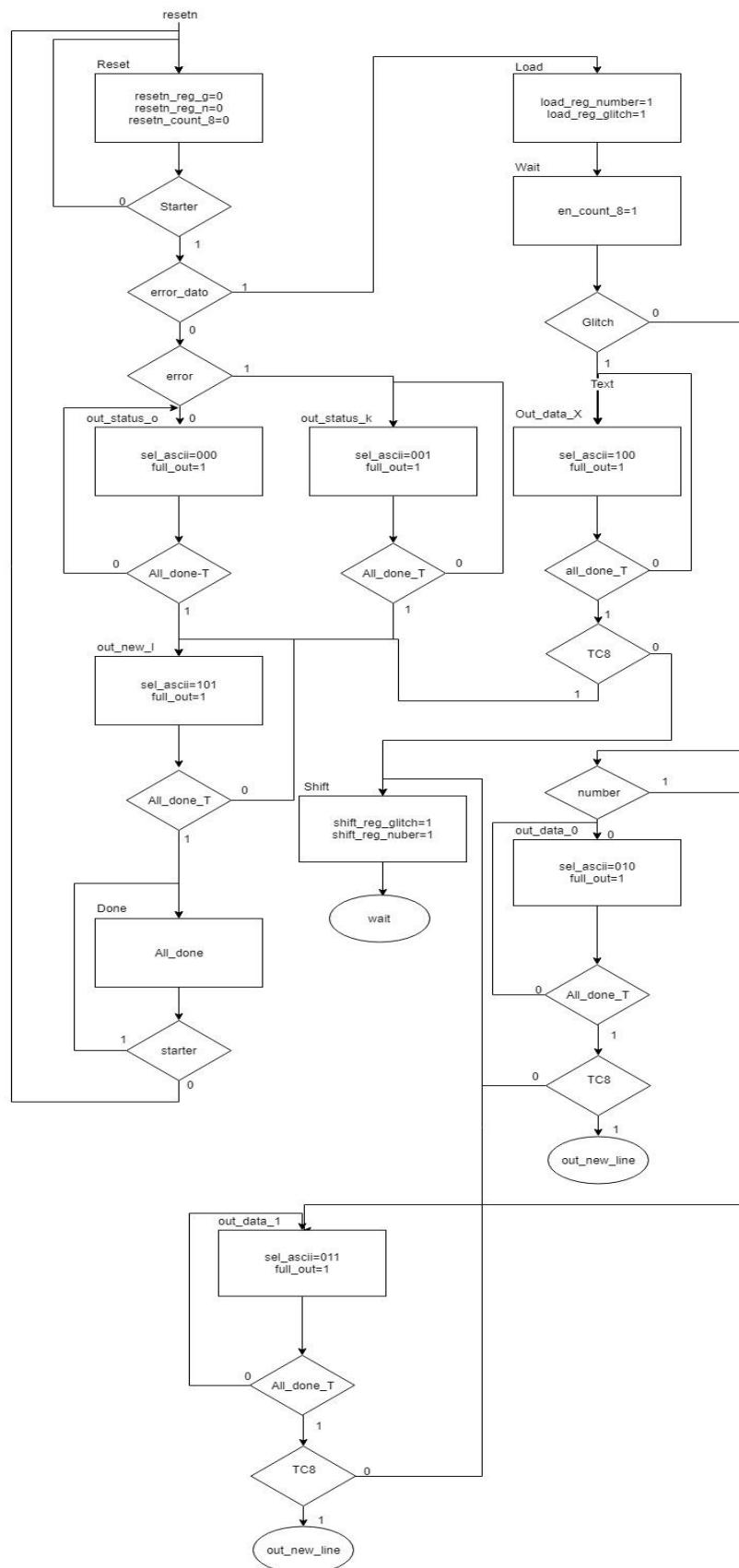


ASM chart



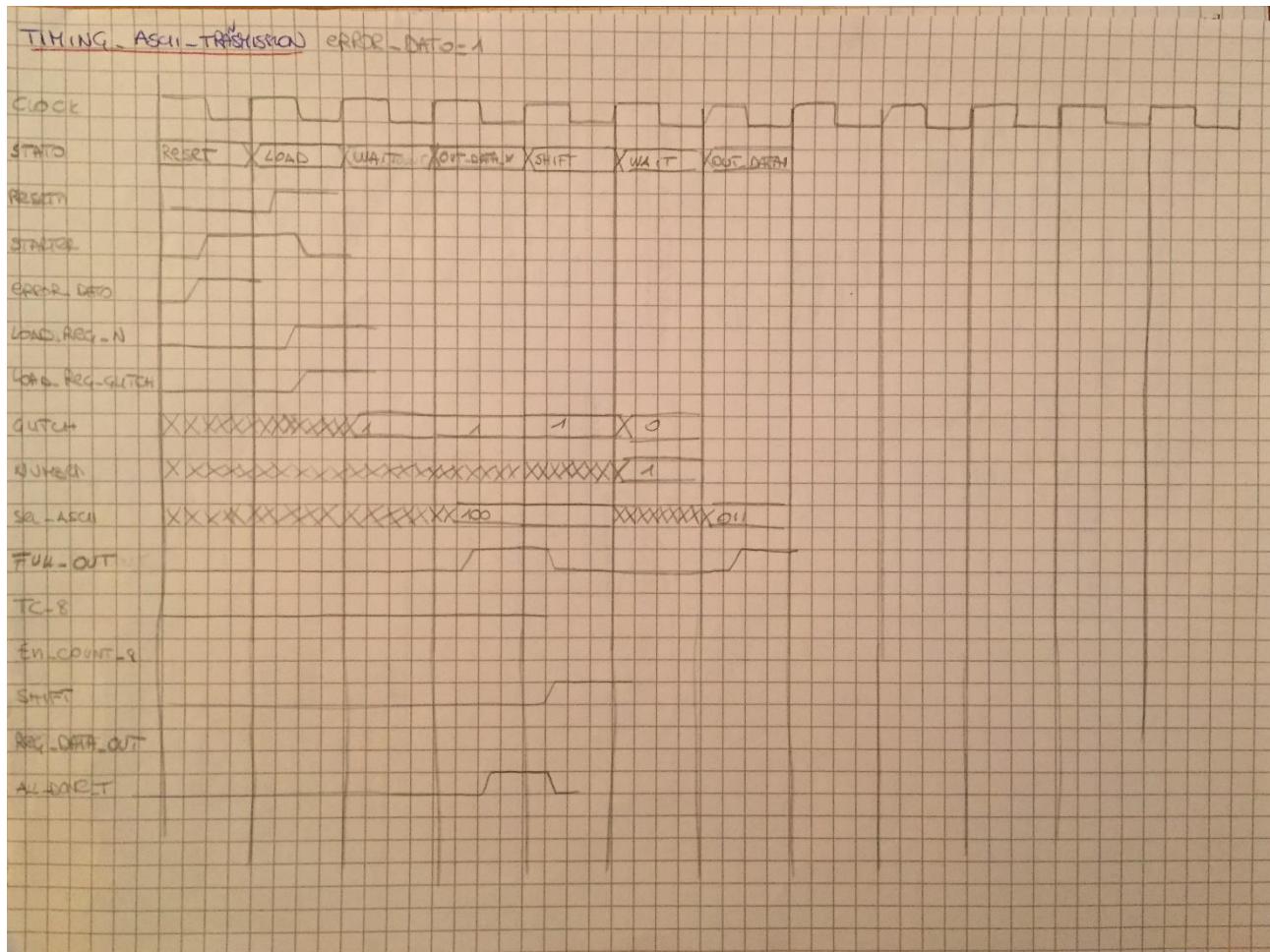
ASM chart dell'ASCII trasmission

Control ASM chart



ASM chart dei controlli dell'ASCII trasmissione

Timing



Timing dell'ASCII trasmissione

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: ASCII_trasmission.vhdl

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY Ascii_transmission IS
PORT( Starter: IN STD_LOGIC;
      Clock,Resetn: IN STD_LOGIC;
      Data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      Glitch_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      error: IN STD_LOGIC;
      error_dato: IN STD_LOGIC;
      all_done_T: IN STD_LOGIC;
      Full_out: OUT STD_LOGIC;
      Data_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```

All_Done: OUT STD_LOGIC);
END Ascii_transmission;
--PORT( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0);
--      KEY:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
--      LEDR: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
--END Acii_transmission;

ARCHITECTURE behavior OF Ascii_transmission IS
--SIGNAL
--SIGNAL starter,clock,resetn,T,All_done: std_logic;
--signal data: std_logic_vector(7 downto 0);
TYPE State_type IS
(Reset,Out_status_o,Out_status_k,Out_new_l,Load,wait_g,Out_data_x,Out_data_0,Out_data_1,Shift,Done);
SIGNAL y : State_type;
SIGNAL
Load_reg_number,Resetn_reg_n,Shift_reg_number,number,glitch,load_reg_glitch,Resetn_reg_g,Shift_reg_glitch,
ch,En_count_8,resetn_count_8,TC_8: STD_LOGIC;
SIGNAL out_count_8 : std_logic_vector(2 downto 0);
SIGNAL sel_ascii: std_logic_vector(2 downto 0);
--COMPONENT
COMPONENT counterupN IS
GENERIC (N : integer:=4);
PORT(   Enable : IN STD_LOGIC;
        Clock , Resetn : IN STD_LOGIC;
        Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END COMPONENT;

```

COMPONENT shift_reg_asynch_reset_8bit is

```

port
(
    clk          : in std_logic;
    enable       : in std_logic;
    resetn      : in std_logic;
    sr_in        : in std_logic;
    pr_in        : in std_logic_vector(7 DOWNTO 0);
    load_parallel : in std_logic;
    sr_out       : out std_logic
);

```

end COMPONENT;

```

COMPONENT mux_6to1_8bit IS
PORT(   A,B,C,D,E,F : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        S : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        Output : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

```

BEGIN

FSM_transitions:

```
PROCESS ( Clock , Resetn )
--VARIABLE first_state : std_logic :='0';
```

BEGIN

```
IF Resetn = '0' THEN
```

```
    y <= Reset;
```

```
ELSIF (Clock 'EVENT AND Clock = '1') THEN
```

```
CASE y IS
```

```
    WHEN Reset => IF starter = '1' THEN IF error_dato='0' THEN IF error='1' THEN y <=
out_status_k;
```

```
        ELSE y <= Out_status_o;
        END IF;
```

```
        ELSE y<=Load;
        END IF;
```

```
        ELSE y<=Reset;
        END IF ;
```

```
    WHEN Out_status_o =>IF All_done_T = '1' THEN y <= Out_new_l ;
```

```
        ELSE y <= Out_status_o;
        END IF ;
```

```
    WHEN Out_new_l =>IF All_done_T = '1' THEN y <= Done ;
```

```
        ELSE y <= Out_new_l;
        END IF ;
```

```
    WHEN Out_status_k =>IF All_done_T = '1' THEN y <= Out_new_l ;
```

```
        ELSE y <= Out_status_k;
        END IF ;
```

```
    WHEN Load => y<=wait_g;
```

```
    WHEN Wait_g =>IF glitch = '0' THEN IF number='1' THEN y <= Out_data_1;
```

```
        ELSE y<=Out_data_0;
        END IF;
```

```
        ELSE y <= Out_data_x;
        END IF ;
```

```
    WHEN Out_data_x =>IF All_done_T = '1' THEN IF TC_8='1' THEN y <= Out_new_l;
```

```
        ELSE y<=Shift;
        END IF;
```

```
        ELSE y <= Out_data_x;
        END IF ;
```

```
    WHEN Out_data_0 =>IF All_done_T = '1' THEN IF TC_8='1' THEN y <= Out_new_l;
```

```
        ELSE y<=Shift;
        END IF;
```

```
        ELSE y <= Out_data_0;
        END IF ;
```

```
    WHEN Out_data_1 =>IF All_done_T = '1' THEN IF TC_8='1' THEN y <= Out_new_l;
```

```
        ELSE y<=Shift;
        END IF;
```

```
        ELSE y <= Out_data_1;
        END IF ;
```

```

WHEN Shift => y<=wait_g;

WHEN Done => IF Starter = '1' THEN y <= Done ;
    ELSE y <= Reset ;
    END IF ;
END CASE;
END IF;
END PROCESS;

```

FSM_outputs: PROCESS(y)

```

BEGIN
resetn_reg_g<='1';
load_reg_glitch<='0';
Shift_reg_glitch<='0';
resetn_reg_n<='1';
load_reg_number<='0';
Shift_reg_number<='0';
Full_out<='0';
en_count_8<='0';
All_Done<='0';
Resetn_count_8<='1';

```

CASE y IS

```

WHEN Reset=> Resetn_reg_g <= '0';
    Resetn_reg_n<='0';
WHEN Out_status_o => Sel_ascii <= "000";
    Full_out<='1';
WHEN Out_new_l => Sel_ascii <= "101";
    Full_out<='1';
WHEN Out_status_k => Sel_ascii <= "001";
    Full_out<='1';
WHEN Load => Load_reg_number<='1';
    Load_reg_glitch<='1';
WHEN wait_g =>resetn_reg_g<='1';
WHEN Out_data_x => Sel_ascii <= "100";
    Full_out<='1';
WHEN Out_data_0 => Sel_ascii <= "010";
    Full_out<='1';
WHEN Out_data_1 => Sel_ascii <= "011";
    Full_out<='1';
WHEN Shift => Shift_reg_glitch<='1';
    Shift_reg_number<='1';
    en_count_8<='1';
WHEN Done => All_Done <= '1';
END CASE;
END PROCESS;

```

Counter_10:counterupN

 GENERIC MAP (N => 3)

```

PORT MAP (en_count_8 , Clock , Resetn_count_8 ,out_count_8);
TC_8 <= out_count_8(0) AND out_count_8(1) AND out_count_8(2);

```

Reg_number: shift_reg_asynch_reset_8bit

```

PORT MAP(Clock,shift_reg_number,Resetn_reg_n,'0',Data_in,load_reg_number,number);

```

Reg_glitch: shift_reg_asynch_reset_8bit

```

PORT MAP(Clock,shift_reg_glitch,Resetn_reg_g,'0',Glitch_in,load_reg_glitch,glitch);

```

Mux: mux_6to1_8bit

```

PORT MAP("01001111","01001011","00110000","00110001","01011000","00001101", sel_ascii,Data_out);

```

```

-- starter<=SW(8);
-- clock<=KEY(1);
-- resetn<=KEY(0);
-- data<=SW(7 downto 0);
-- LEDR(0)<=T;
-- LEDR(1)<=All_Done;
END behavior;

```

UART transmitter



Blocco della UART transmitter

Il dato da inviare viene mandato su “Data”.

Un registro a 10 bit viene caricato nel seguente modo:

in Reg_T(0)->0 bit di start

in reg_T(8 downto 1)->Dato da inviare

in reg_T(9)-> 1 bit di guardia

La linea è impostata con un bit rate a 115200. Ogni bit dunque è inviato per 434 colpi di clock (clock a 50 MHz).

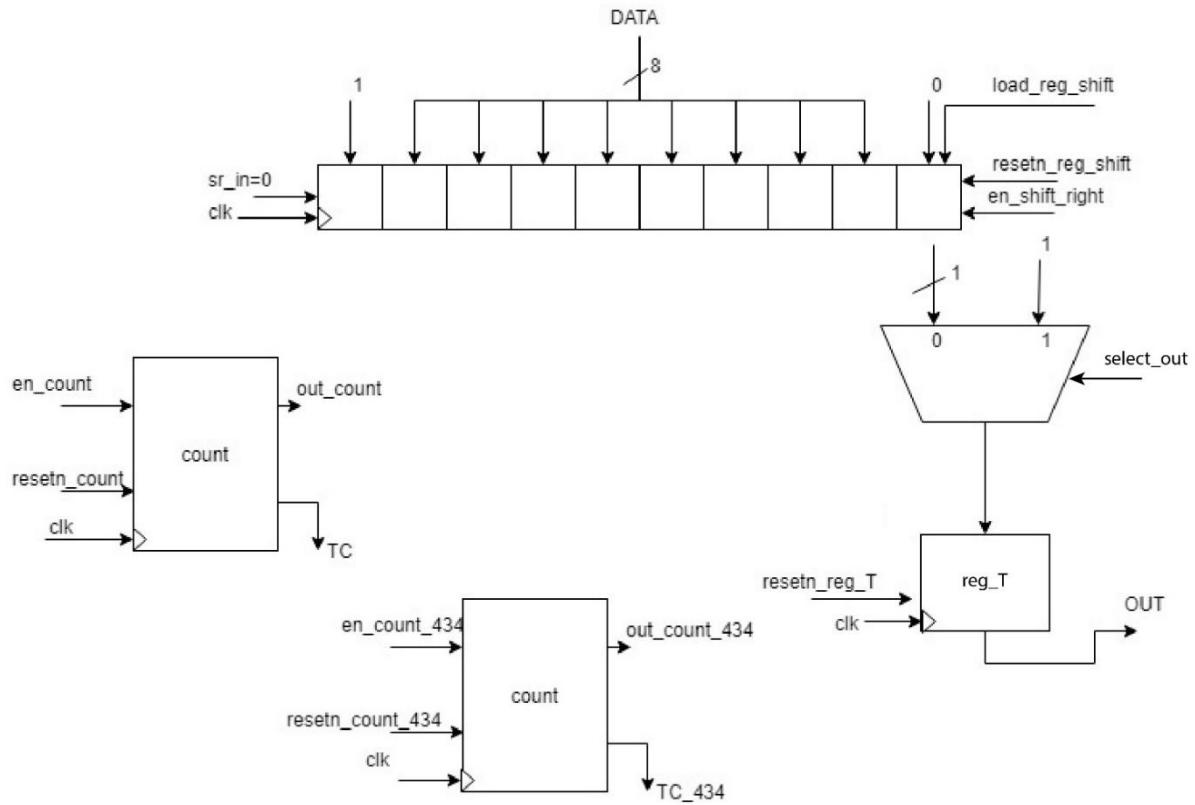
Un contatore count434 conta i colpi di clock per ogni singolo bit. Il count10 tiene traccia dei 10 bit da inviare.

Una volta finito il dato da inviare (TC10) la linea invierà, fino al prossimo start, il bit= ‘1’.

Pseudocodice

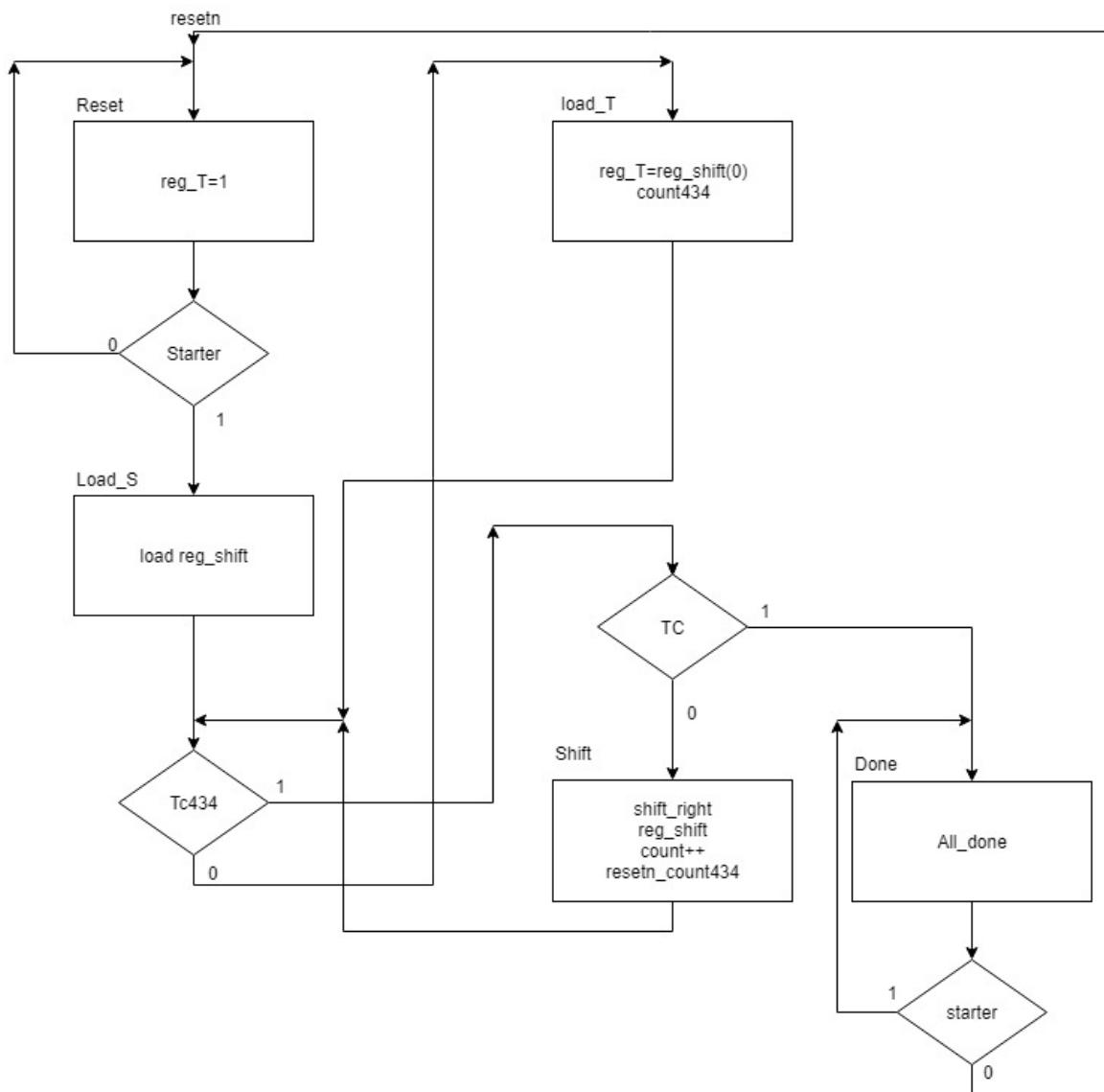
```
Reg_shift=0;  
Reg_T=1;  
If start=1  
    reg_shift(0)=0; reg_shift(9)=1;  
    reg_shift(8 downto 0)= data(7 downto 0);  
    for i=0:9  
        for j=0:434  
            reg_T=reg_shift(0); out=reg_T;  
        end_for;  
        shift reg_shift;  
    end for;  
end if;
```

Datapath



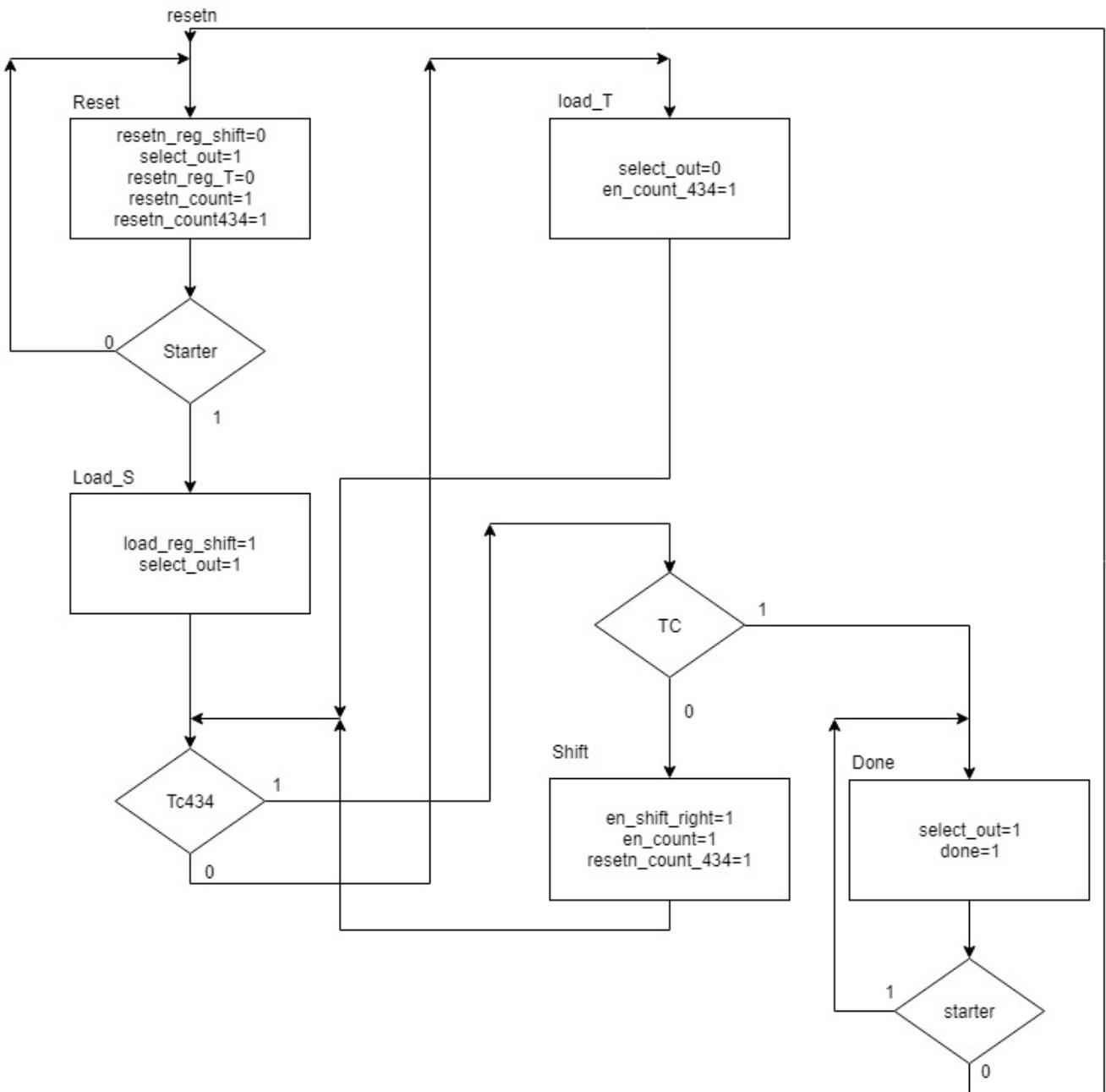
Datapath della UART transmitter

ASM chart



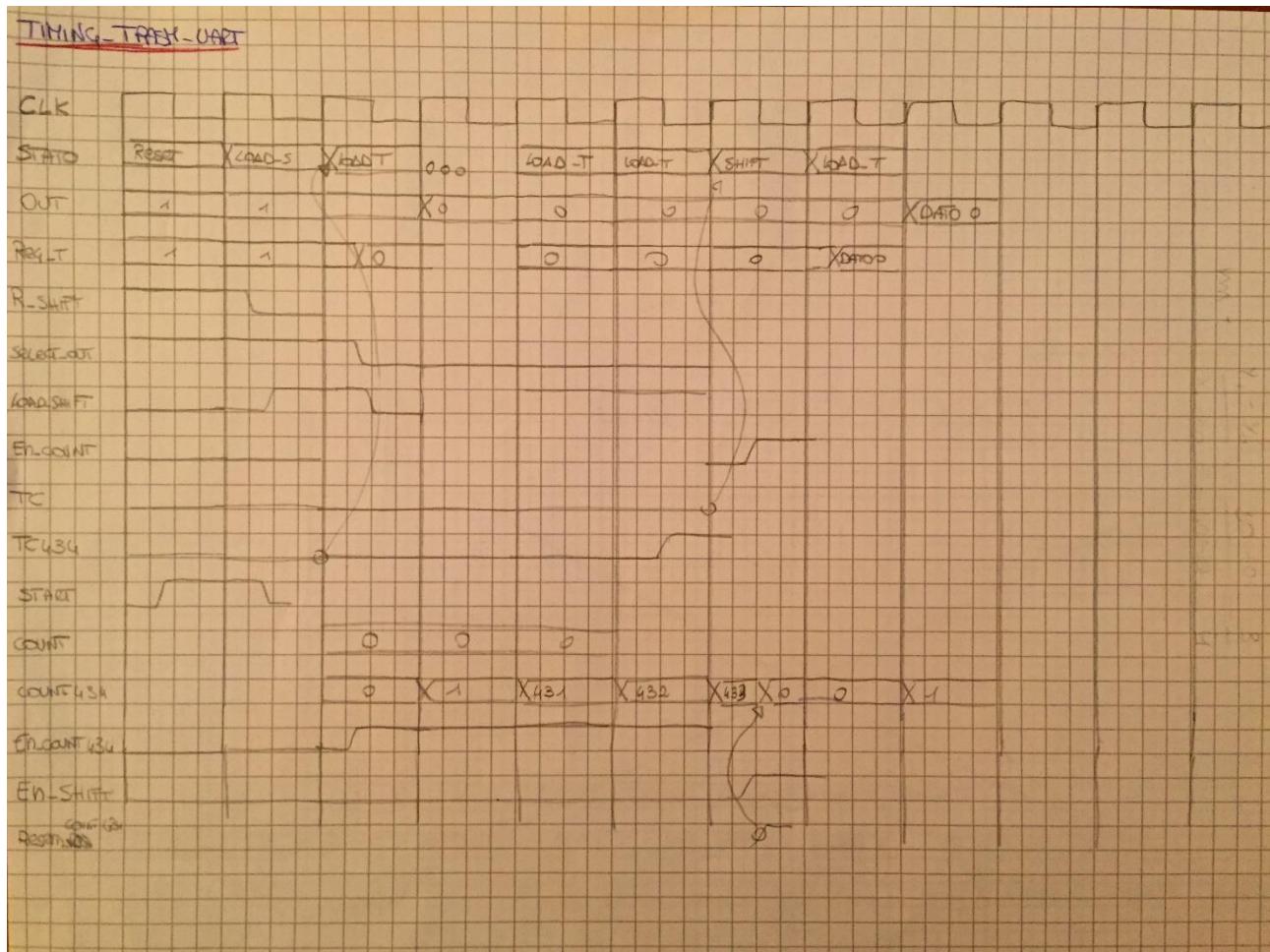
ASM chart della UART transmitter

Control ASM chart



ASM chart dei controlli della UART transmitter

Timing



Timing della UART transmitter

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: UART_transmitter.vhdl

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;

ENTITY trasmettitore IS
PORT( Starter: IN STD_LOGIC;
      Clock,Resetn:IN STD_LOGIC;
      Data: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      T: OUT STD_LOGIC;
      All_Done: OUT STD_LOGIC);
END trasmettitore;
-PORT( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0);

```

```
-- CLOCK_50: IN STD_LOGIC;
-- UART_TXD: OUT STD_LOGIC;
-- LEDR: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
-END trasmettitore;
```

ARCHITECTURE behavior OF trasmettitore IS

-SIGNAL

```
--SIGNAL starter,clock,resetn,T,All_done: std_logic;
```

```
--SIGNAL data: std_logic_vector(7 downto 0);
```

```
TYPE State_type IS (Reset,Load_S,Load_T,Shift,Done);
```

SIGNAL y : State_type;

SIGNAL out_shift,TC, in_reg_T, en_shift_right, load_reg_shift, select_out,

en_count,resetn_reg_shift,resetn_reg_T,resetn_count,en_count_434,tc_434,resetn_count_434: STD_LOGIC;

SIGNAL out_count : std_logic_vector(3 downto 0);

SIGNAL out_count_434:std_logic_vector(8 downto 0);

-COMPONENT

COMPONENT counterupN IS

GENERIC (N : integer:=4);

PORT(Enable : IN STD_LOGIC;

Clock , Resetn : IN STD_LOGIC;

Output : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));

END COMPONENT;

COMPONENT shift_reg_asynch_reset is

```
port
(
    clk      : in std_logic;
    enable   : in std_logic;
    resetn   : in std_logic;
    sr_in    : in std_logic;
    pr_in    : in std_logic_vector(7 DOWNTO 0);
    load_parallel : in std_logic;
    sr_out   : out std_logic
);
```

end COMPONENT;

COMPONENT flipflopD IS

PORT (

D, Clock , Resetn : IN STD_LOGIC;

Q :OUT STD_LOGIC);

END COMPONENT;

COMPONENT mux2to1 IS -- dichiarazione entita '

PORT (x, y, s : IN STD_LOGIC;

m : OUT STD_LOGIC);

END COMPONENT;

```

BEGIN

FSM_transitions:
PROCESS ( Clock , Resetn )
--VARIABLE first_state : std_logic :='0';

BEGIN
IF Resetn = '0' THEN
  y <= Reset;
ELSIF (Clock 'EVENT AND Clock = '1') THEN
  CASE y IS
    WHEN Reset => IF starter = '1' THEN y <= Load_S ;
      ELSE y <= Reset ;
      END IF ;
    WHEN Load_S => IF TC_434 = '1' THEN IF TC='1' THEN y <= DONE ;
      ELSE y <= Shift ;
      END IF ;
      ELSE y<=load_T;
      END IF;
    WHEN Load_T => IF TC_434 = '1' THEN IF TC='1' THEN y <= DONE ;
      ELSE y <= Shift ;
      END IF ;
      ELSE y<=load_T;
      END IF;
    WHEN Shift => IF TC_434 = '1' THEN IF TC='1' THEN y <= DONE ;
      ELSE y <= Shift ;
      END IF ;
      ELSE y<=load_T;
      END IF;
    WHEN Done => IF Starter = '1' THEN y <= Done ;
      ELSE y <= Reset ;
      END IF ;
      END CASE;
    END IF;
END PROCESS;

```

```

FSM_outputs: PROCESS(y)
BEGIN
resetn_reg_shift<='1';
resetn_reg_T<='1';
select_out<='1';
Load_reg_shift<='0';
en_count<='0';
en_shift_right<='0';
All_Done<='0';
Resetn_count<='1';
en_count_434<='0';
Resetn_count_434<='1';

```

```

CASE y IS
WHEN Reset=> Resetn_reg_shift <= '0';
    Resetn_count<='0';
    Resetn_count_434<='0';
    Select_out<='1';
WHEN Load_S => Load_reg_shift <= '1';
    Select_out<='1';
WHEN Load_T => en_count_434 <= '1';
    Select_out<='0';
WHEN Shift =>en_shift_right<='1';
    en_count<='1';
    Resetn_count_434<='0';
WHEN Done => All_Done <= '1';
END CASE;
END PROCESS;

```

Counter_10:counterupN

```

GENERIC MAP (N => 4)
PORT MAP (en_count , Clock , Resetn_count ,out_count);
TC <= NOT out_count(0) AND out_count(1) AND NOT out_count(2) AND out_count(3);

```

Reg_shift: shift_reg_asynch_reset

```

PORT MAP(Clock,En_shift_right,Resetn_reg_shift,'1',Data(7 downto 0),load_reg_shift,out_shift);

```

Mux: mux2to1

```

PORT MAP(out_shift,'1', select_out,in_reg_T);

```

Reg_T: flipflopD

```

PORT MAP(in_Reg_T,clock,Resetn_Reg_T,T);

```

Counter_434:counterupN

```

GENERIC MAP (N => 9)
PORT MAP (en_count_434 , Clock , Resetn_count_434 ,out_count_434);
TC_434 <= NOT out_count_434(0) AND out_count_434(1) AND NOT out_count_434(2) AND NOT
out_count_434(3)AND out_count_434(4) AND out_count_434(5) AND NOT out_count_434(6) AND
out_count_434(7) AND out_count_434(8);

```

```

-- starter<=SW(8);
-- resetn<=SW(17);
-- CLOCK<=CLOCK_50;
-- data<=SW(7 downto 0);
-- UART_RXD<=T;
-- LEDR(1)<=All_Done;
END behavior;

```

UART receiver

Il ricevitore riceve i dati dal pc 1 bit alla volta tramite interfaccia asincrona UART e poichè il nostro clock è a 50 Mhz e i dati vengono trasmessi a 115200 KHz si è scelto di abilitare un contatore ogni volta il clock arrivasce a 115200 KHz in modo da prendere il dato con la temporizzazione corretta. La macchina una volta beccato lo start bit comincia a contare per mettersi a metà dello start bit (conta fino a 216), terminata questa operazione la macchina campiona il primo bit a metà circa (in modo da evitare letture errate). Prosegue in questo modo prendendo gli 8 bit (in questo caso il contatore a 512 bit conta fino a 430 mettendo a circa metà dei successivi bit) finchè dopo aver preso gli 8 bit si mette a metà della linea per il controllo dello stop bit e in seguito arriva al DONE dando ACK se la trasmissione dei dati è avvenuta in modo corretto e quindi si è beccato lo stop bit.

Pseudocodice

IF START=1

FF0=r;

IF FF0=0

DELAY(4 CLK)

FF0= R;

END IF;

IF FF0=0

FOR I=0:7

DELAY(8 CLK);

SHIFT_REG(I)=FF0;

END FOR;

END IF;

DELAY(8 CLK);

IF FF0=1

ACK=0;

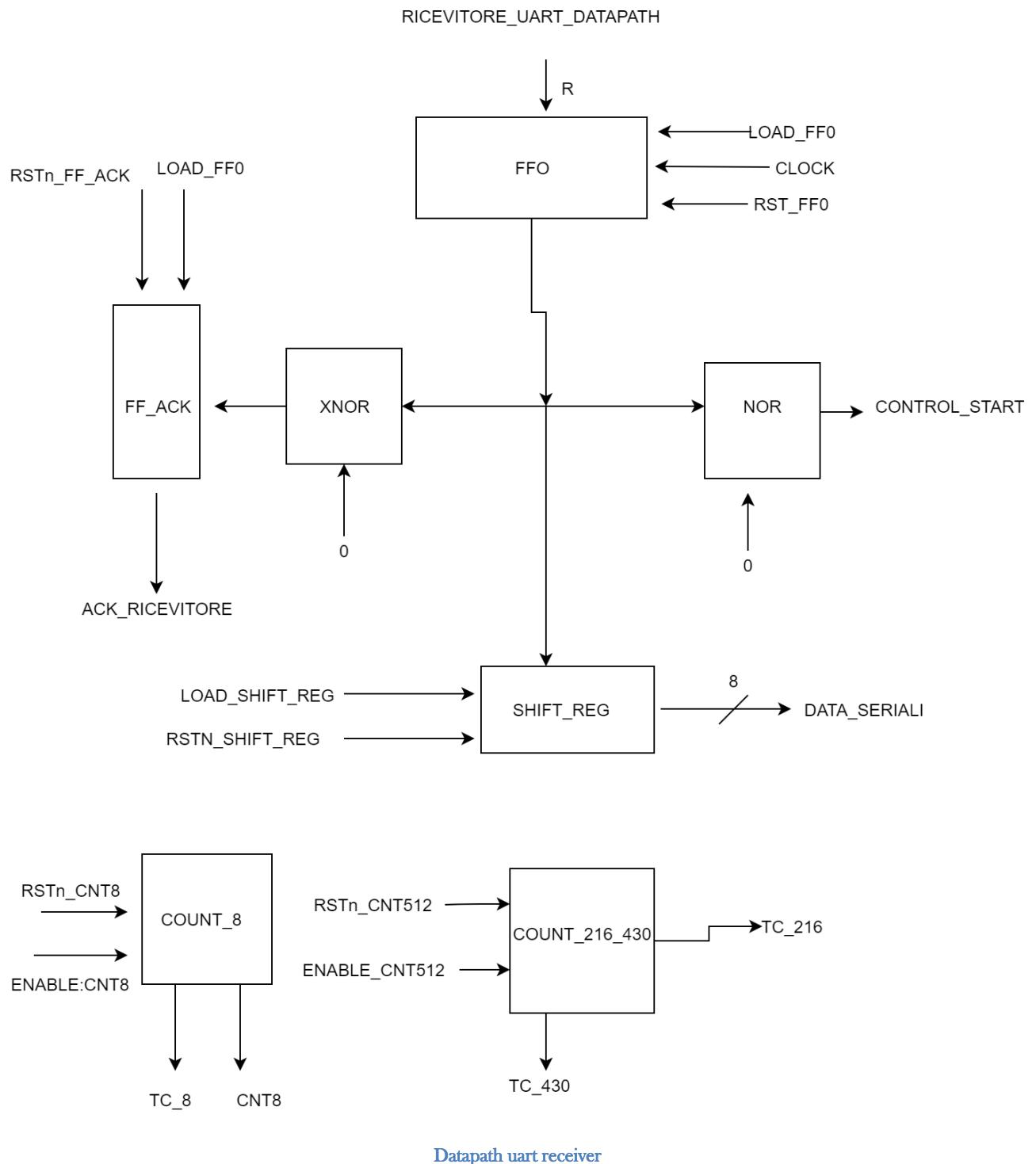
ELSE

ACK=1;

ENDIF;

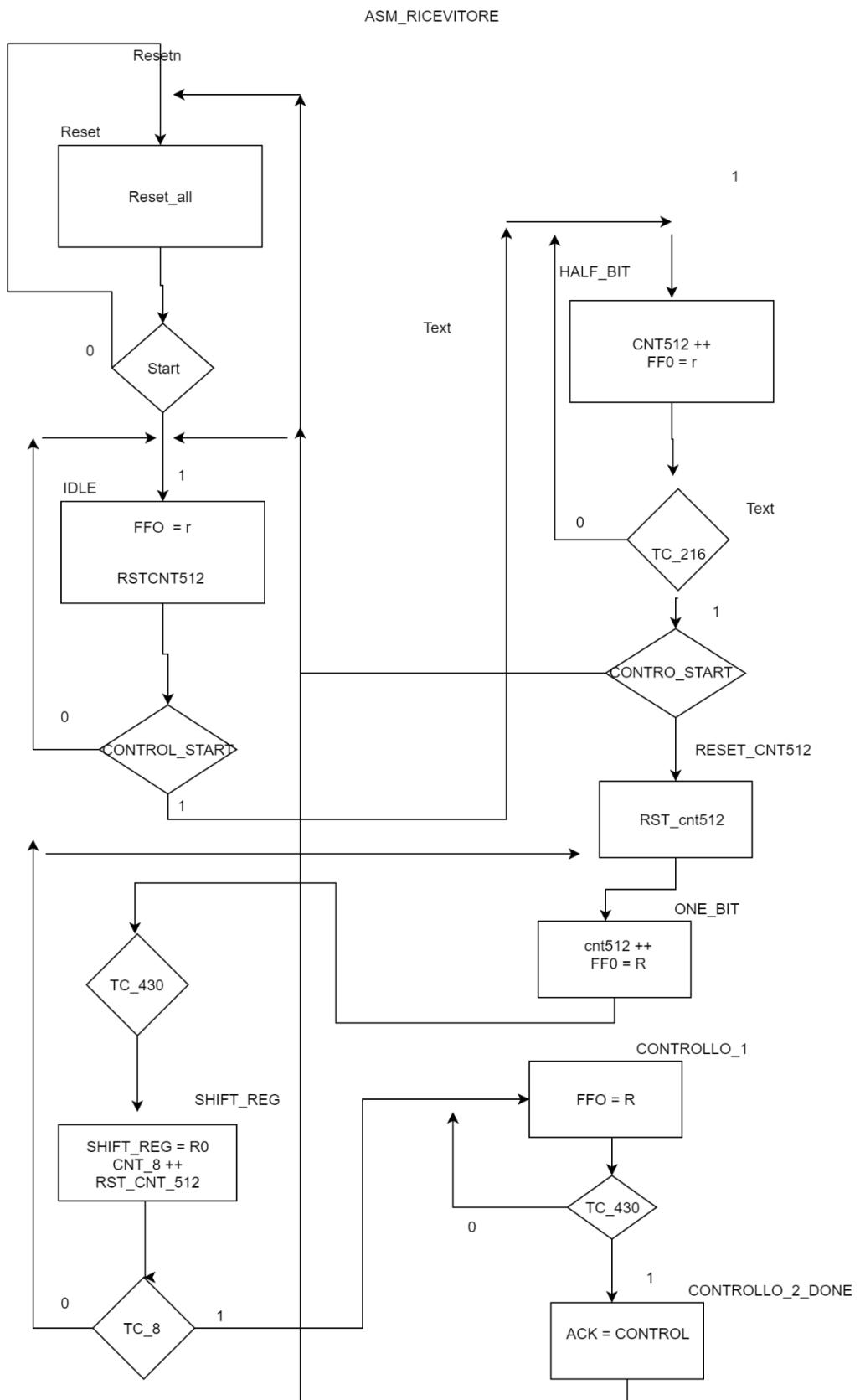
ENDIF;

Datapath



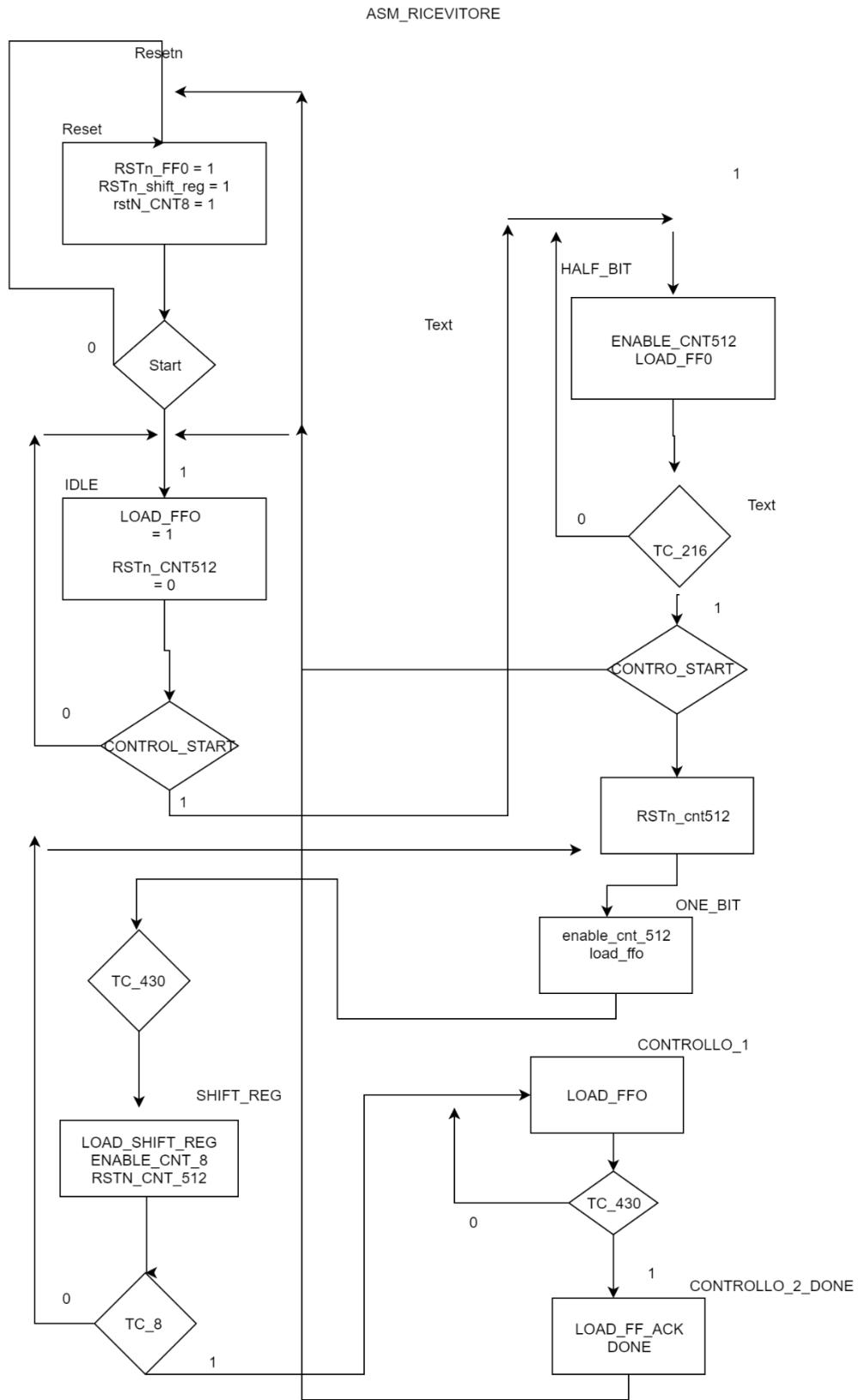
Datapath uart receiver

ASM chart



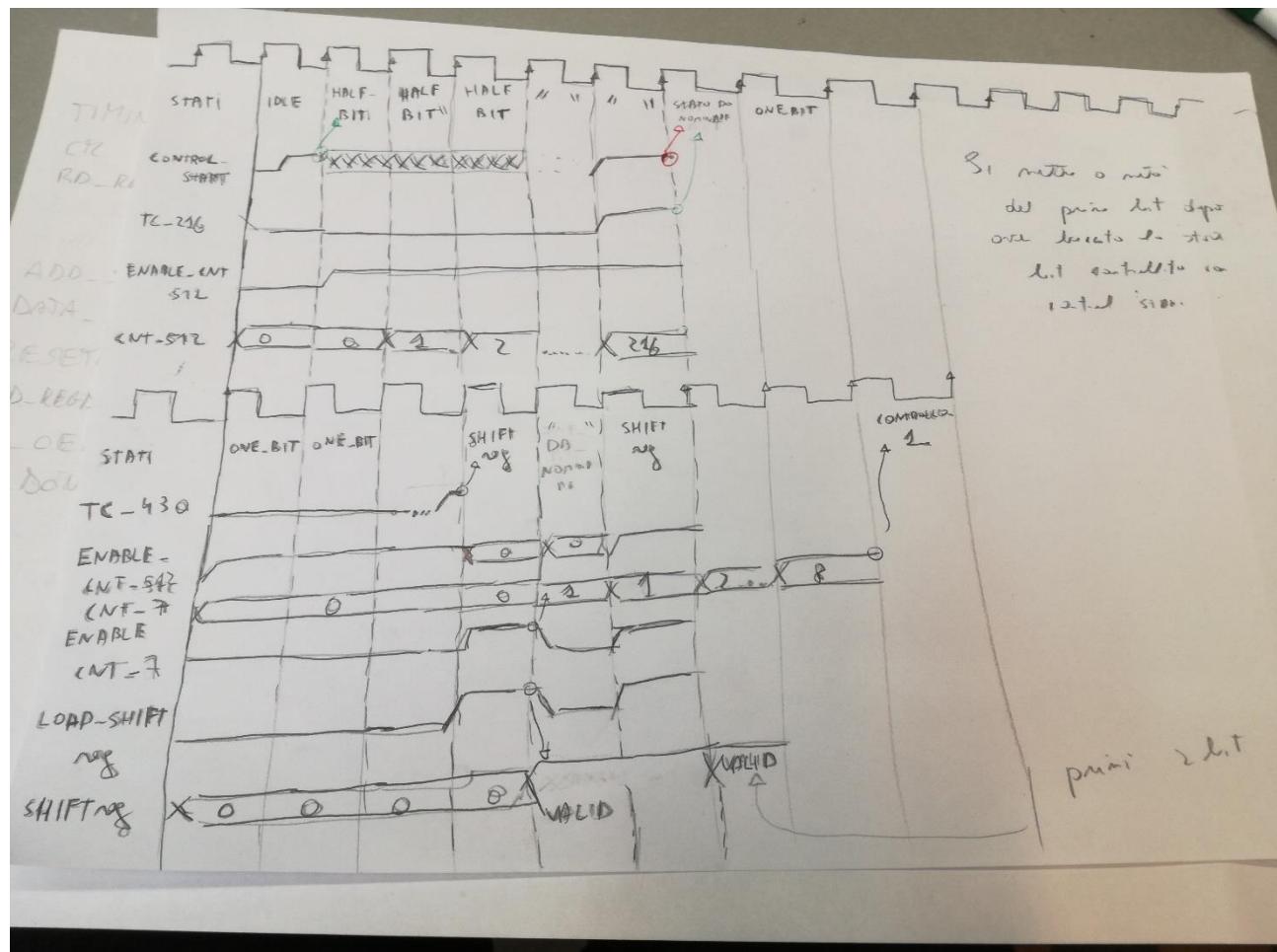
ASM chart del uart receiver

Control ASM chart



ASM chart dei controlli del uart receiver

Timing



Timing del uart receiver

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II.

File VHDL: **UART_receiver.vhdl**

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
USE ieee.std_logic_unsigned.all ;
```

```
ENTITY uart_ricevitore IS
```

```

PORT ( start_ricevitore_uart: IN STD_LOGIC;
clock,resetN_uart_ricevitore: IN STD_LOGIC;      --SONO DUE SEGNALI DI START E RESET SOLO
DELLA RICEVITORE NON QUELLO GENERALE, MANDATO DALLA C.U generale
r: IN STD_LOGIC;
done : OUT STD_LOGIC;
data_seriali : out std_logic_vector ( 7 downto 0);
ack : OUT STD_LOGIC );
END uart_ricevitore;

```

ARCHITECTURE BEHAVIOUR OF uart_ricevitore IS

```

component count512 IS
--GENERIC ( N : INTEGER:=8 ) ;
PORT ( ResetN : IN STD_LOGIC ;
carico_dato : IN STD_LOGIC_VECTOR ( 8 DOWNTO 0);
E, Clock , LOAD_CNT : IN STD_LOGIC;
TC : OUT STD_LOGIC;          --216
tcl : out std_logic;        --430
Q : OUT STD_LOGIC_VECTOR (8 DOWNTO 0) );
END component;

```

```

COMPONENT count7 IS
--GENERIC ( N : INTEGER:=8 ) ;

```

```

PORT ( ResetN : IN STD_LOGIC ;
carico_dato : in std_logic_vector ( 2 downto 0 );
E, Clock, LOAD_CNT : IN STD_LOGIC ;
TC, tc1 : OUT STD_LOGIC;
Q : OUT STD_LOGIC_VECTOR (2 DOWNTO 0) ) ;
END COMPONENT;

```

```

component flipflopsr IS
PORT ( D, Clock, resetn, LOAD_FF : IN STD_LOGIC;
set_reset : IN STD_LOGIC;
Q : OUT STD_LOGIC ) ;
END component ;

```

```

COMPONENT shift8 IS
PORT ( w, Clock, load_shift, ResetN : IN STD_LOGIC ;
Q : OUT STD_LOGIC_VECTOR(7 downto 0));
END COMPONENT;

```

```

TYPE state_type is (RESET_RICEVITORE, IDLE, HALF_BITE, STATO_INTERMEDIO, ONE_BITE,
SHIFT_REG, CONTROLLO_1, CONTROLLO_2, DONE_RICEVITORE);

```

```

SIGNAL STATE : state_type;
```

```

SIGNAL rstN_ff_ack , load_shift_reg , control_start, r0 , control , enable_cnt7, enable_cnt8 ,load_ff_ack,
load_ff0, rstN_ff0, rstN_shift_reg, rstN_cnt8, TC_8, TC_8_1 : std_logic;
```

```

SIGNAL CNT8 : std_logic_vector (2 DOWNTO 0);
```

```
--signal start_ricevitore_uart , clock, resetn_uart_ricevitore , done , ack , r : std_logic;  
--signal data_seriali : std_logic_vector ( 7 downto 0 );
```

```
--signal rst_cnt3472, enable_cnt3472 : std_logic;
```

```
signal rstN_cnt512 , enable_cnt512, load_cnt512, tc_216, tc_430 : std_logic ;  
signal out_cnt_512 : std_logic_vector ( 8 downto 0 );
```

BEGIN

```
--start_ricevitore_uart <= sw(0);  
--clock <= clock_50;  
--resetN_uart_ricevitore <= sw(1);  
--r <= UART_RXD;
```

FSM_transitions: PROCESS (resetN_uart_ricevitore, clock)

BEGIN

IF (resetN_uart_ricevitore= '0') THEN

STATE <=RESET_RICEVITORE; -RESET ASINCRONO

ELSIF (Clock'EVENT AND Clock = '1') THEN

CASE STATE IS

WHEN RESET_RICEVITORE => IF start_ricevitore_uart = '1' THEN STATE<=IDLE ; ELSE STATE <= RESET_RICEVITORE ; END IF ;

WHEN IDLE => IF control_start = '1' THEN STATE <= HALF_BITE ; ELSE STATE <= IDLE ; END IF ;

WHEN HALF_BITE => --IF tc_7 = '0' THEN STATE<=HALF_BITE; ELSE IF control_start = '0' then STATE<=IDLE; ELSE STATE <= ONE_BITE; END IF; END IF;

if tc_216 = '0' then state <= HALF_BITE; ELSE IF control_start = '0' then STATE<=IDLE; ELSE STATE <= STATO_INTERMEDIO; END IF; END IF;

WHEN STATO_INTERMEDIO => STATE <= ONE_BITE;

WHEN ONE_BITE => --IF tc_7_2 = '0' THEN STATE <= ONE_BITE; ELSE STATE <= SHIFT_REG; END IF;

if tc_430 = '0' then state <= one_bite; else state <= shift_reg; end if;

WHEN SHIFT_REG => IF tc_8 = '0' THEN STATE <= STATO_INTERMEDIO; ELSE STATE <= CONTROLLO_1; END IF; -- controllare

WHEN CONTROLLO_1 => --IF tc_7 = '1' THEN STATE <= CONTROLLO_2; ELSE STATE <= CONTROLLO_1; END IF;

```
if tc_430 = '0' then state <= controllo_2; else state <=
controllo_1; end if;
```

```
WHEN CONTROLLO_2 => STATE <= DONE_RICEVITORE;
```

```
WHEN DONE_RICEVITORE => IF start_ricevitore_uart = '1' then STATE <= DONE_RICEVITORE ;
ELSE STATE <= RESET_RICEVITORE; END IF;
```

```
WHEN OTHERS => STATE <= RESET_RICEVITORE; -- controllo
```

```
END CASE ;
```

```
END IF ;
```

```
END PROCESS ;
```

```
FSM_OUTPUT : PROCESS (STATE) IS
```

```
BEGIN
```

```
enable_cnt8 <= '0'; load_ff_ack <= '0'; load_ff0 <= '0'; rstN_ff0 <='1'; rstN_shift_reg <='1'; rstN_cnt8 <= '1';
```

```
rstN_ff_ack <= '1'; load_shift_reg <= '0'; done <= '0'; rstN_CNT512 <= '1'; enable_cnt512 <= '0';
```

```
load_cnt512 <= '0';
```

```
CASE STATE is
```

```
WHEN RESET_RICEVITORE => rstN_ff0 <='0'; rstN_shift_reg <='0'; rstN_cnt8 <= '0'; rstN_ff_ack <= '0';
rstN_CNT512 <= '0'; --rstN_CNT7 <= '0';
```

```
WHEN IDLE => load_ff0 <= '1'; rstN_cnt512 <= '0' ;
```

```
WHEN HALF_BITE => enable_cnt512 <= '1'; load_ff0 <= '1';
```

```
when stat0_intermedio => rstN_cnt512 <= '0' ;
```

```
WHEN ONE_BITE => enable_cnt512 <= '1'; load_ff0 <= '1';
```

```
WHEN SHIFT_REG => enable_cnt8 <= '1'; load_shift_reg <= '1'; rstN_CNT512 <= '0' ;
```

```
WHEN CONTROLLO_1 => load_ff0 <= '1'; enable_cnt512 <= '1';
```

```
WHEN CONTROLLO_2 => Load_ff_ack <= '1';
```

```
WHEN DONE_RICEVITORE => Done <= '1';
```

```
END CASE;
```

```
END PROCESS;
```

```
--datapath
```

```
-ledr(8) <= done ;
```

```
-ledr(7 downto 0) <= data_seriali;
```

```
-ledr(9) <= ack;
```

```
SHIFT_REG1: shift8 port map ( r0, clock, load_shift_reg ,rstN_shift_reg,data_seriali);
```

```
FF0 : flipflopsr port map (r, clock, rstN_FF0 ,load_ff0,'1' , r0 );
```

--vedere cosa mettere

bene ovvero se nel FF ci vuole il reset

```
--COUNT_4 : COUNT7 PORT MAP (rstN_CNT7, "100" , enable_cnt7, clock, load_4, tc_7, tc_7_2 , CNT7 );
```

```
--COUNT_5 : COUNT7 PORT MAP ('1', "000" , enable_cnt7, clock, load_1, tc_7 , CNT7 );
```

--nel circuito di vuole un mux che sceglie su cosa caricare

```
COUNT_8 : COUNT7 PORT MAP (rstN_cnt8 , "000" , enable_cnt8, clock , '0' , tc_8, tc_8_1 , CNT8);
```

```
ACKNOWLEDGE : flipflopsr port map (control, clock, rstN_FF_ACK, LOAD_FF_ACK, '0', ACK);
```

```
count_216_430 : count512 port map ( rstN_cnt512, "000000000" , enable_cnt512, clock , load_cnt512, tc_216, tc_430, out_cnt_512 );
```

```
--Gen_3472 : countN port map (rst_cnt3472 , enable_cnt3472, clock , conteggio_3472);
```

```
control <= ('0' Xnor r0);
```

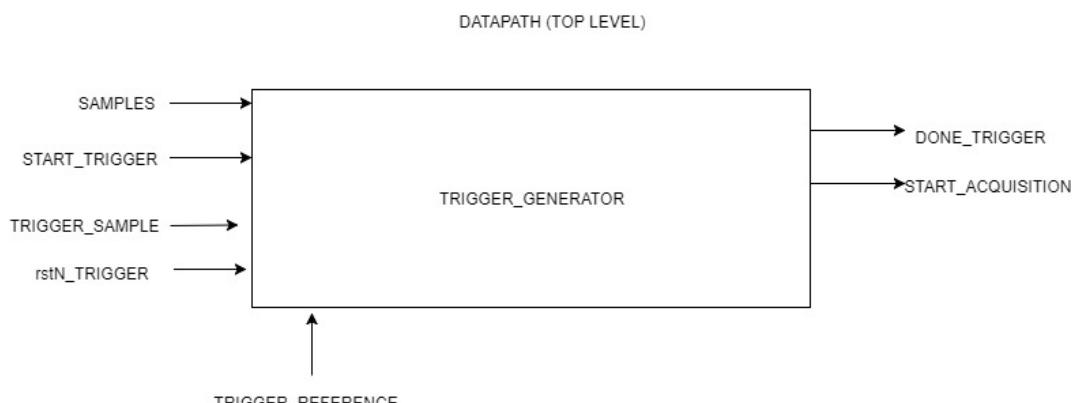
```
control_start <= '0' nor r0 ;
```

```
end behaviour;
```

Trigger generator

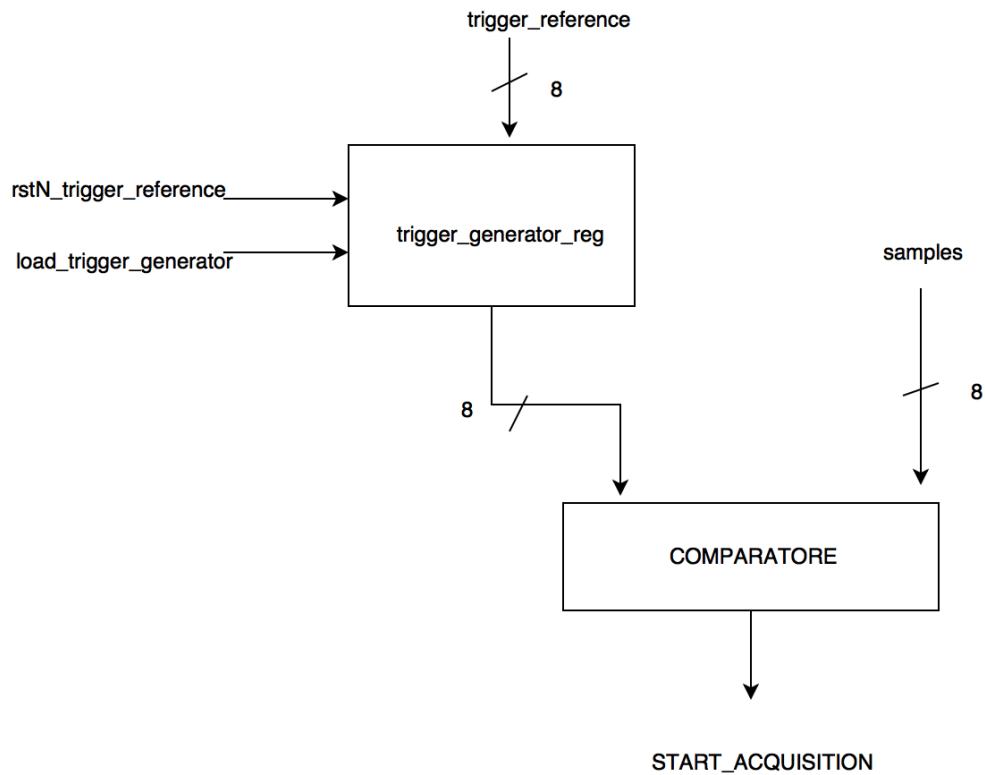
Il trigger è un blocco quasi puramente combinatorio dove ogni colpo di clock si cerca l'hit tra i dati provenienti dal ricevitore e i dati campionati, dopo essersi verificato l'hit del trigger viene mandato un segnale che arriva alla MAIN e permette che venga campionato il futuro riempendo la seconda parte della memoria tenendo presente l'indirizzo di inizio del futuro che poi sarà necessario in lettura.

Il blocco è sintetizzabile come segue.



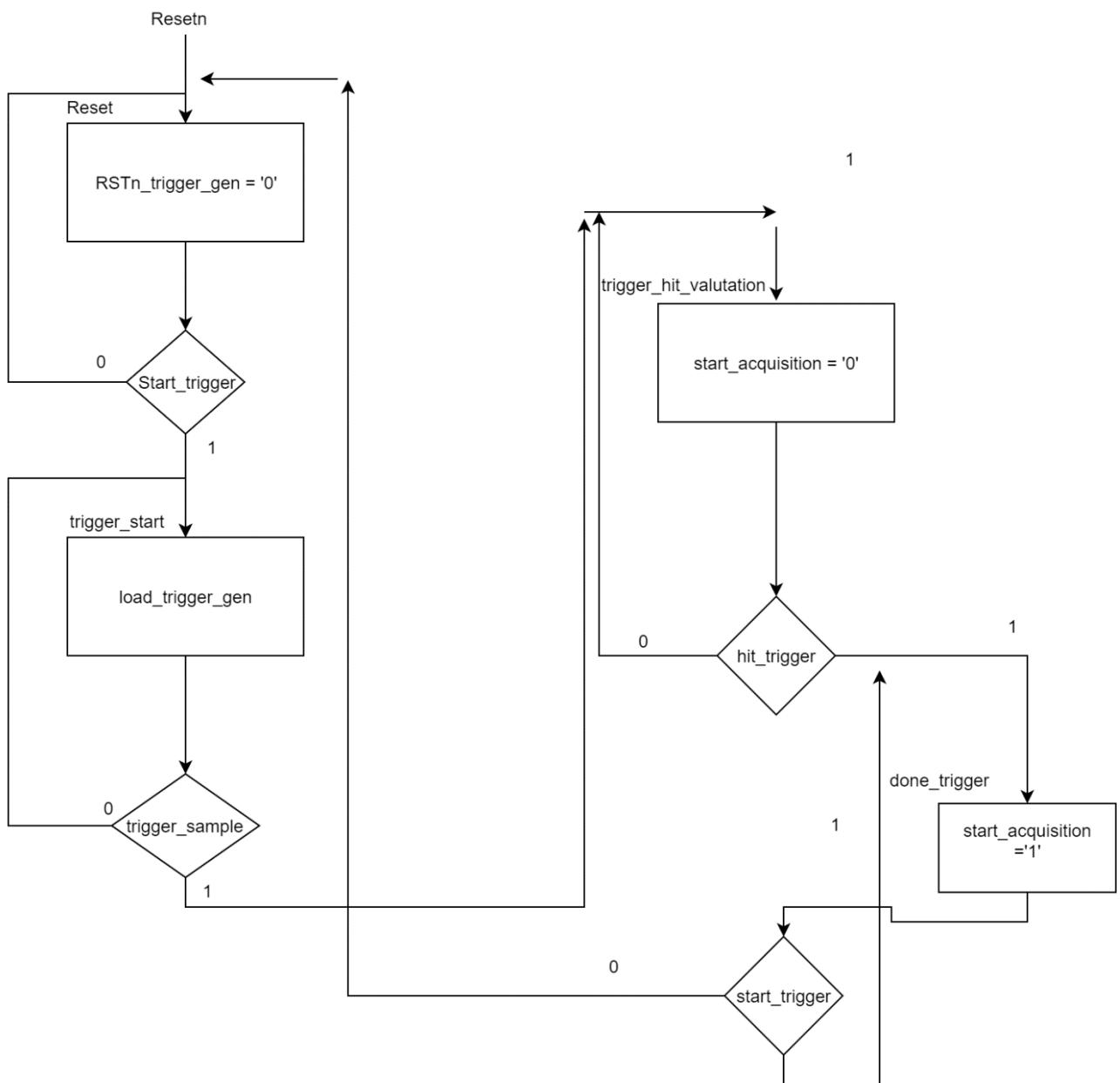
Blocco di generazione del segnale di trigger

Datapath



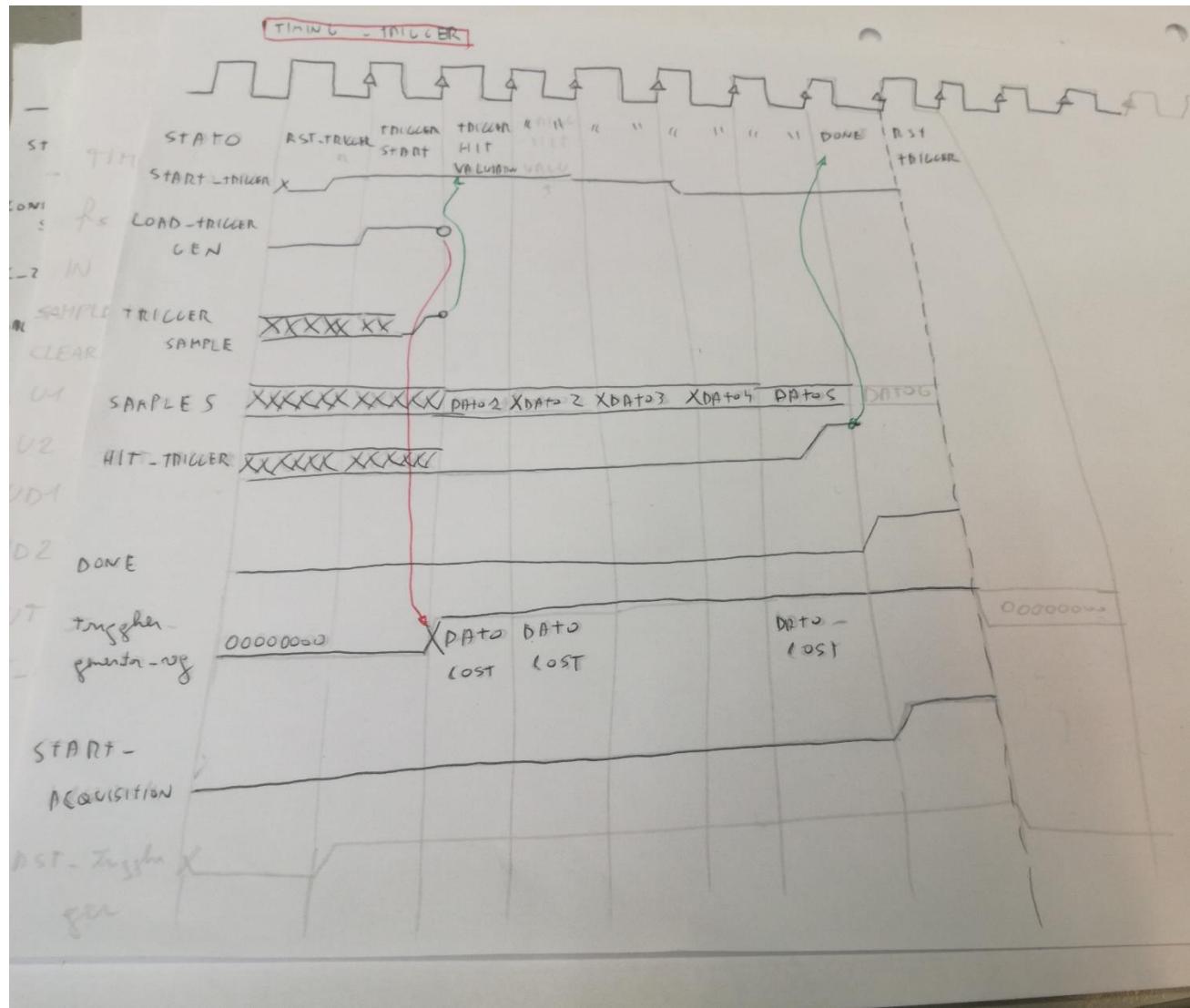
Datapath del blocco di generazione del segnale di trigger

Control ASM chart



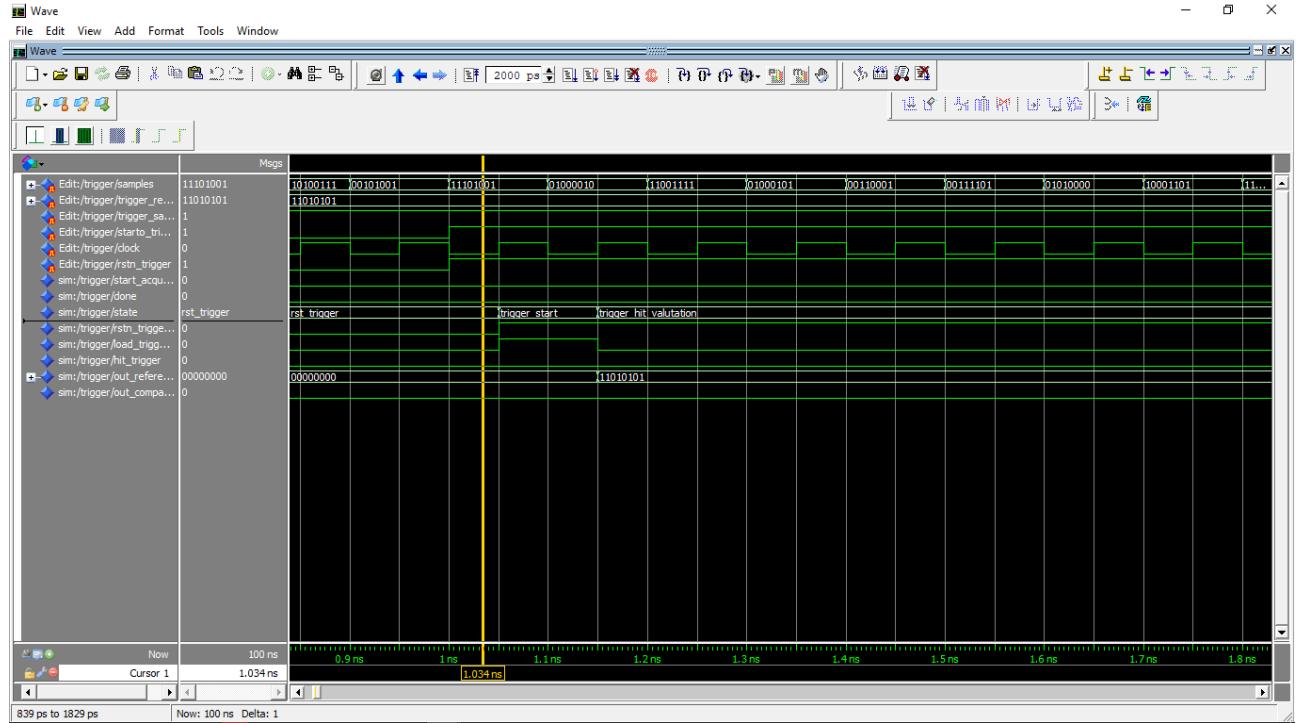
Control ASM chart del blocco di generazione del segnale di trigger

Timing



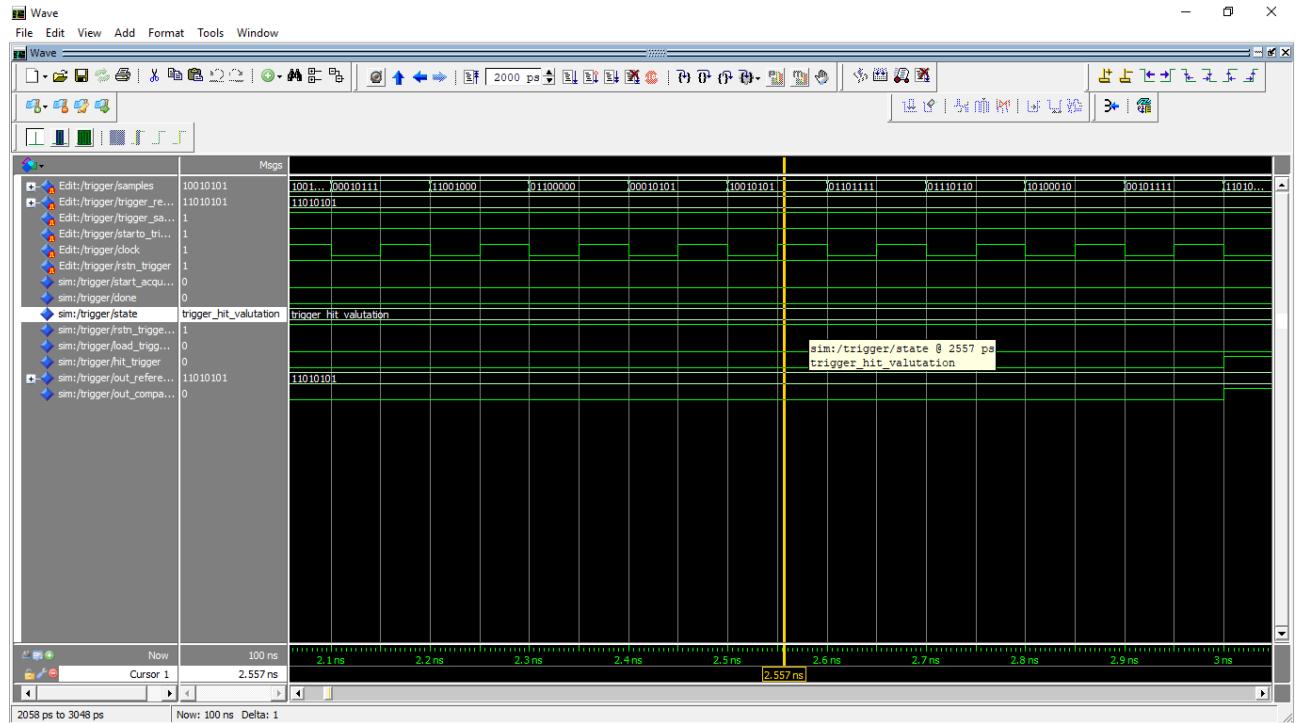
Simulazioni

Il trigger è stato simulato in 3 situazioni notevoli. Dopo lo start_trigger, la macchina passa dallo stato di reset_trigger allo stato di trigger_start (stato di idle della macchina).



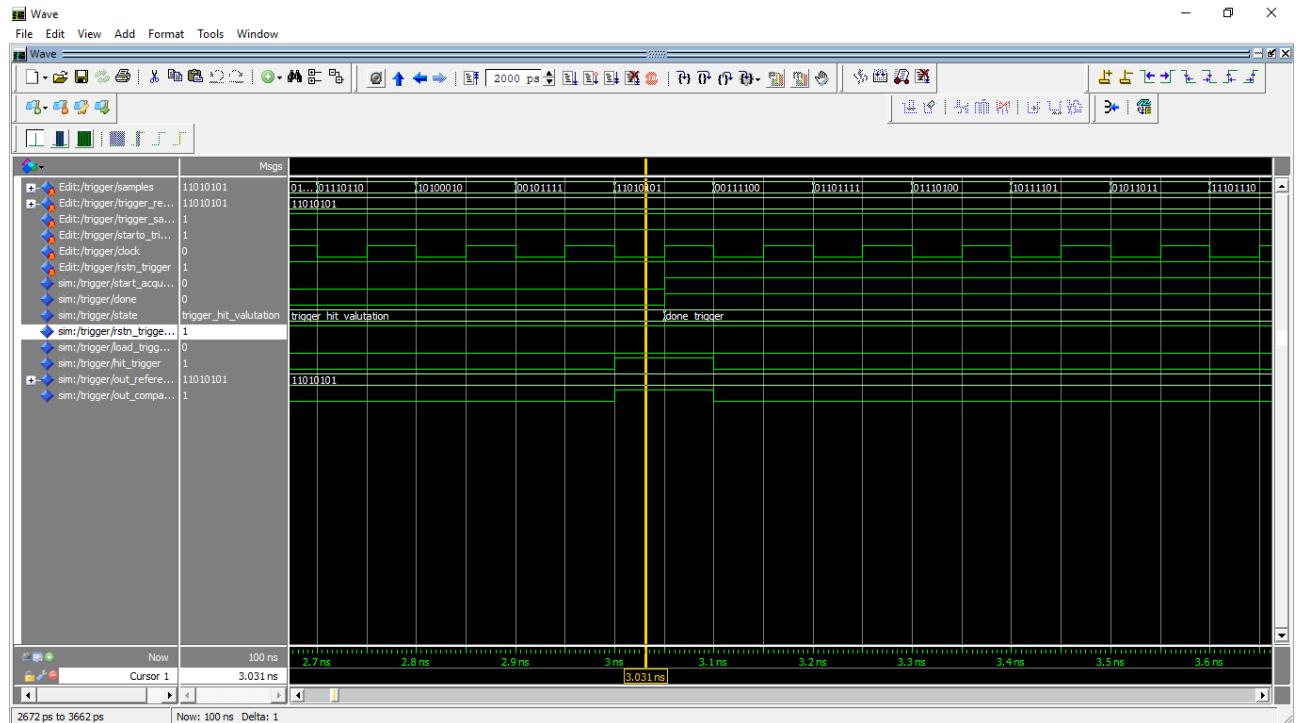
Prima simulazione del blocco di generazione del segnale di trigger

Nella seconda simulazione siamo nello stato di trigger_hit_evaluation dove il trigger confronta i segnali campionati con il trigger_reference cercando l'uguaglianza tra i dati confrontati.



Seconda simulazione del blocco di generazione del segnale di trigger

Nell'ultima simulazione, la macchina passa dallo stato di trigger_hit_evaluation allo stato di DONE dopo aver triggerato i campioni.



Terza simulazione del blocco di generazione del segnale di trigger

Codice VHDL

NOTA: la stesura del progetto è stata fatta usando il linguaggio VHDL, il compilatore QUARTUS II e la verifica del funzionamento (simulazioni) con il simulatore MODELSIM.

File VHDL : trigger.vhdl

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
entity trigger is
Port ( trigger_reference : IN STD_LOGIC_VECTOR (7 downto 0);
samples :IN STD_LOGIC_VECTOR (7 downto 0);
trigger_sample, starto_trigger, Clock : in STD_LOGIC;
RSTn_TRIGGER : IN STD_LOGIC;
start_acquisition : OUT std_logic; -- BUFFER: both IN & OUT
done : OUT STD_LOGIC);
```

```
end trigger;
```

architecture Behaviour of trigger is

```

COMPONENT reg8 IS
    PORT ( D : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
        Resetn, Clock, load_reg : IN STD_LOGIC ;
        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END COMPONENT;

```

```

COMPONENT comparator_8bit IS
    PORT ( A,B: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        U: OUT STD_LOGIC);
END COMPONENT;

```

```

-- name of all states
type state_type IS ( RST_TRIGGER, TRIGGER_START, TRIGGER_HIT_VALUTATION,
DONE_TRIGGER );
SIGNAL STATE : STATE_TYPE;
-SIGNAL INTERNAL
SIGNAL RSTn_TRIGGER_GEN : STD_LOGIC;
SIGNAL LOAD_TRIGGER_GEN, HIT_TRIGGER : STD_LOGIC;
SIGNAL OUT_REFERENCE : STD_LOGIC_VECTOR(7 DOWNTO 0); -- filo in uscita del trigger
generator
SIGNAL OUT_COMPARATORE : STD_LOGIC; -- filo in uscita dal comparatore

```

```
BEGIN
```

```
FSM_TRANSITIONS : PROCESS ( rstN_TRIGGER, CLOCK ) -SENSITIVITY LIST
```

```
BEGIN
```

```
IF rstN_TRIGGER = '0' THEN STATE <= RST_TRIGGER;
```

```
ELSIF (Clock'EVENT AND Clock = '1') THEN
```

```
CASE STATE IS
```

```
WHEN RST_TRIGGER => IF starto_trigger = '1' then state <= trigger_start ; else state <= RST_TRIGGER;
end if;
```

```
WHEN TRIGGER_START => IF trigger_sample = '1' then state <= TRIGGER_HIT_VALUTATION; else
state <= trigger_start ; end if;
```

```
WHEN TRIGGER_HIT_VALUTATION => IF HIT_TRIGGER = '1' then state <= DONE_TRIGGER ;
else state <= TRIGGER_HIT_VALUTATION ; end if;
```

```
when DONE_TRIGGER => if starto_trigger = '1' then state <= DONE_TRIGGER; else state <= rst_trigger;
end if;
```

when others => state <= RST_TRIGGER; **-PER EVENTUALI STATI NON DEFINITI E PREVEDERNE IL PERCORSO**

end CASE;

end if;

end PROCESS;

FSM_OUTPUT : PROCESS (STATE) --EVEN IN IF MEALY MACHINE

BEGIN

-DEFAULT signal for datapath

RSTn_TRIGGER_GEN <= '1';

LOAD_TRIGGER_GEN <= '0';

DONE <= '0';

START_ACQUISITION <= '0';

CASE state IS

WHEN RST_TRIGGER => RSTn_TRIGGER_GEN <= '0';

WHEN TRIGGER_START => LOAD_TRIGGER_GEN <= '1';

WHEN TRIGGER_HIT_VALUTATION => START_ACQUISITION <= '0';

WHEN DONE_TRIGGER => DONE <= '1'; START_ACQUISITION <= '1'; **-- lo start acquisition entra in gioco il colpo di clock successivo al campione che è stato triggerato che sarà il primo campione del "futuro"**

END CASE;

END PROCESS;

-DATAPATH

TRIGGER_GEN_REG : reg8 PORT MAP (trigger_reference, RSTn_TRIGGER_GEN, CLOCK, LOAD_TRIGGER_GEN, OUT_REFERENCE);

COMPARATORE : comparator_8bit PORT MAP (OUT_REFERENCE, SAMPLES, OUT_COMPARATORE);

HIT_TRIGGER <= '1' AND OUT_COMPARATORE;

END Behaviour;