Progetto “GPU”

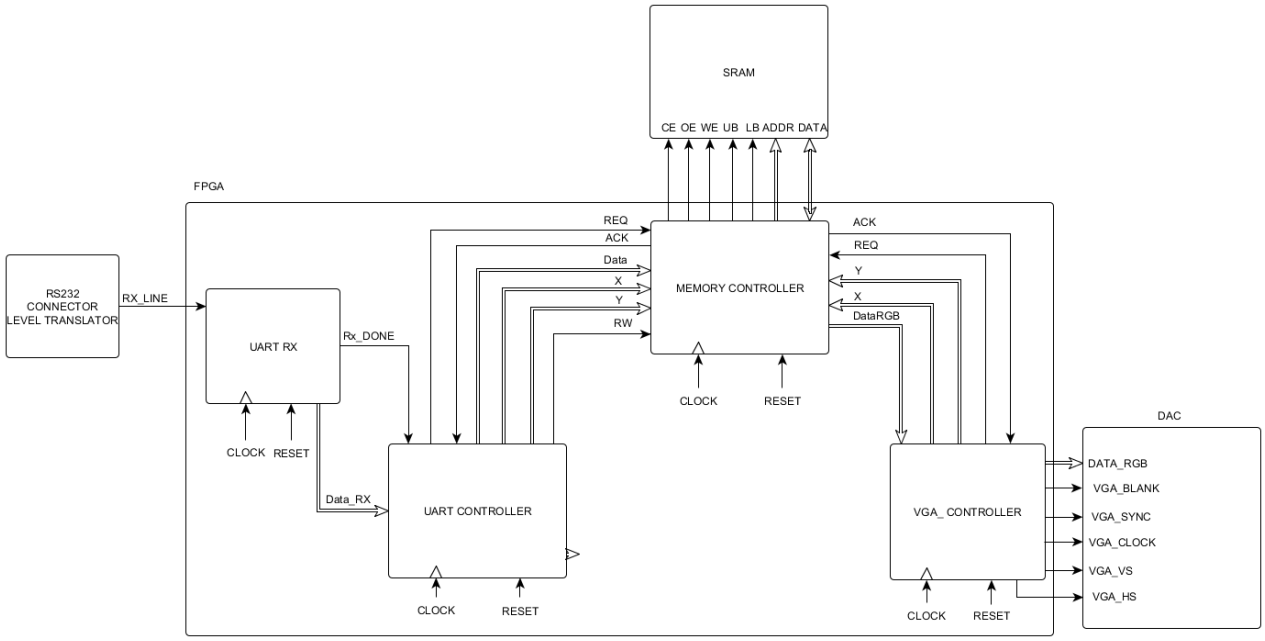
Autori:

Fabio Grassi, Alessandro Nadal, Stefano Iuliano.

Il progetto “GPU” prevede la realizzazione di un sistema digitale - analogico in grado di rappresentare un immagine inviata dall’utente, attraverso un computer, su un monitor. L’invio dei dati avviene grazie alla linea RS232 mentre la rappresentazione utilizza il protocollo VGA. L’unità di elaborazione hardware è rappresentata dalla scheda DE2 Altera. Essa contiene al suo interno una memoria SRAM, un FPGA, un convertitore DAC e un traslatore di livello RS232. L’insieme di questi blocchi permette di realizzare il sistema seguente.

Nota: Anche se all’interno del progetto viene usato il solo ricevitore UART è stato richiesto anche il progetto del trasmettitore.

Sistema



Di seguito sono mostrate le fasi di progetto di ogni blocco.

1.0 Blocco UART

Il blocco UART provvede alla gestione di una linea seriale RS232 full-duplex con velocità di trasmissione di 9600 bps. Esso è composto internamente da due macchine a stati, una per il trasmettitore e una per il ricevitore.

1.1 Trasmettitore RS232

Il trasmettitore RS232 provvede, in seguito ad una richiesta proveniente dall'esterno, ad inviare un dato a 8 bit attraverso la linea seriale. Il dato è preceduto da un bit di START. La trasmissione termina correttamente se viene inviato il bit di STOP.

1.1.1 Specifiche

La linea seriale in assenza di trasmissione è a livello logico '1', IDLE. Non appena il trasmettitore inizio l'invio di un dato forza a livello logico '0' la linea, inviando così il bit di Start. Successivamente vengono inviati gli 8 bit del dato, partendo dal meno significativo. La trasmissione termina con il bit di STOP a livello logico '1'. Il trasmettitore quindi può iniziare una nuova trasmissione con un segnale di START oppure lasciare in IDLE la linea.
Avendo deciso di trasmettere ad una velocità di 9600 bps, la durata di ogni bit è pari a 1/9600=104 µs. La base tempi utilizzata è un clock a 25 MHz. È pertanto necessario utilizzare un contatore per ottenere la velocità di trasmissione di 9600 bps. Di seguito è presentato lo pseudo codice dell'algoritmo per realizzare la macchina a stati del trasmettitore.

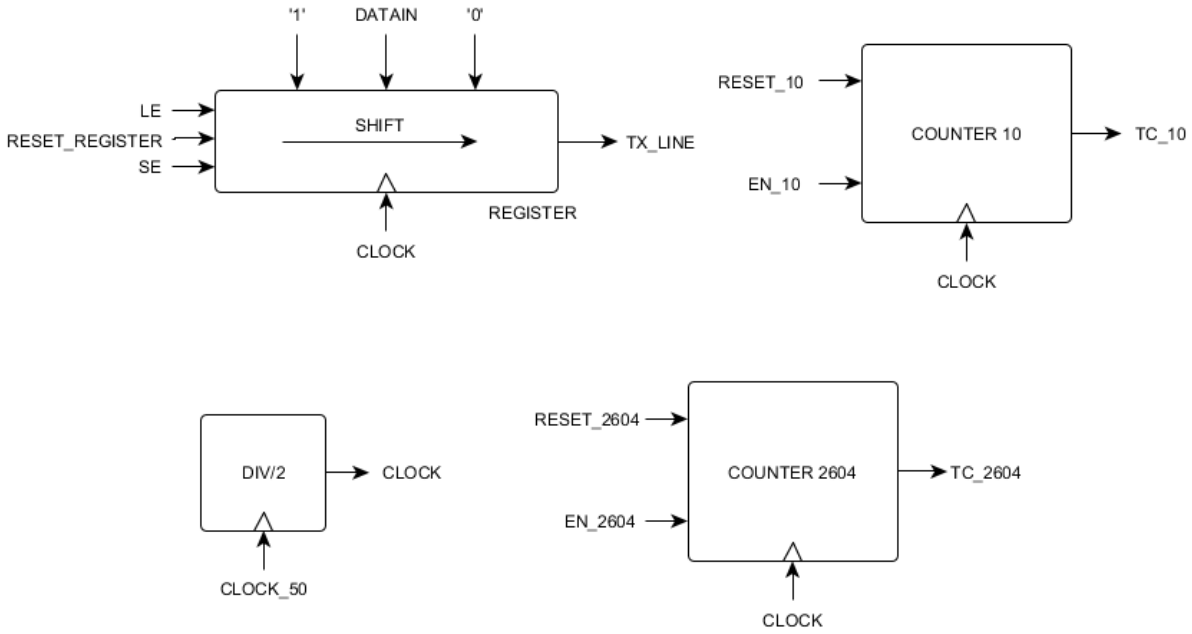
1.1.2 Pseudo Codice

```
REGISTER="1111111111";
COUNT_10=0;
COUNT_2604=0;
TX_DONE=1;
if(TX_REQ=1){
TX_DONE=0;
REGISTER='1' & DATAIN & '0';
FOR(;COUNT_10<=9;COUNT_10++){
TX_LINE=SHIFT(REGISTER);
FOR(COUNT_2604=0;COUNT_2604;COUNT_2604++);
}
TX_DONE=1;
```

Il dato inviato dal trasmettitore è contenuto nel registro REGISTER che inizialmente contiene "1111111111". Questa sequenza corrisponde allo stato IDLE della linea. Successivamente vengono azzerati COUNT_10 e COUNT_2604. Il primo è utile per contare il numero di bit inviati, nel nostro caso 10. Il secondo invece viene utilizzato per contare il numero di colpi di clock per garantire 9600 bps.

Se la richiesta di trasmissione TX_REQ è verificata, il registro di trasmissione viene caricato con il dato fornito dall'utente, preceduto dal bit di START e terminato da quello di STOP. In seguito viene azzerato il segnale di uscita TX_DONE, il quale comunica all'utilizzatore che il trasmettitore è occupato nella trasmissione. Vengono poi inviati in successione i bit garantendo la temporizzazione corretta di 9600 bps, ossia mantenendo valido il dato per 2604 colpi di clock. Terminata la trasmissione viene portata a '1' logico l'uscita TX_DONE.

1.1.3 Data Path



- **Register**: costituisce il blocco di memorizzazione del dato fornito dall'utente. Esso permette anche di generare in uscita la sequenza di bit utili all'invio del frame START+DATO+STOP;

- **Counter_10**: contatore modulo 10 utile a contare il numero di bit inviati dal trasmettitore;

- **Counter_2604**: contatore modulo 2604, utilizzato per garantire una velocità di trasmissione dati di 9600 bps;

- **DIV/2**: blocco che permette di ottenere un clock di 25 MHZ attraverso un divisore per due;

Register

Il registro **Register** è uno shift register con caricamento parallelo e parallelismo 10 bit. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencate le porte di I/O ed i segnali di controllo:

- RESET_REGISTER: segnale di reset che permette forzare il contenuto del registro REGISTER a “1111111111”;
- LE: segnale che permette di abilitare il caricamento sincrono del dato presente sulla porta DATA_IN portando l’MSB del registro al valore 1 e l’LSB al valore ‘0’;
- SE: segnale che permette di abilitare lo shift sincrono del dato presente all’interno del registro REGISTER, esso viene inviato in uscita attraverso la linea TX_LINE;
- DATA_IN: porta di ingresso a 8 bit per il caricamento del dato proveniente dall’esterno;
- TX_LINE: segnale di uscita del registro che permette l’invio dei dati seriali;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Counter_10

Il contatore **Counter_10** è un contatore modulo 10. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di controllo:

- RESET_10: segnale di reset che permette di azzerare il contatore portando il segnale TC_10 al valore logico ‘0’;
- EN_10: segnale di ingresso sincrono per abilitare il contatore;
- TC_10: segnale di uscita Terminal Count. Esso rimane attivo per un colpo di clock non appena il contatore raggiunge il valore 9;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Counter_2604

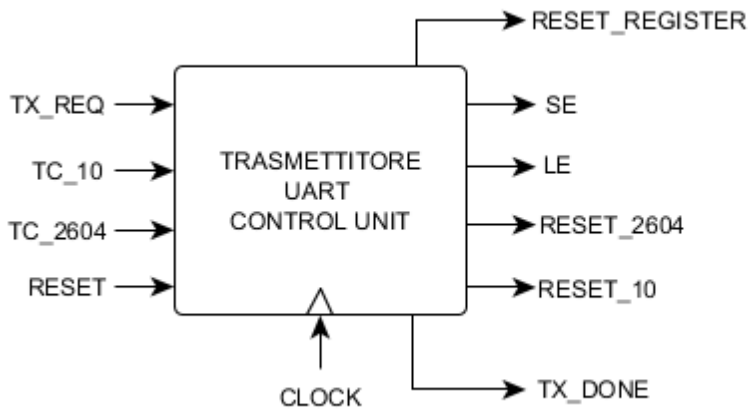
Il contatore **Counter_2604** è un contatore modulo 2604. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RESET_2604: segnale di reset che permette di azzerare il contatore portando il segnale TC_2604 al valore logico ‘0’;
- EN_2604: segnale di ingresso sincrono di abilitazione del contatore;

- TC_2604: segnale di uscita Terminal Count. Esso rimane attivo per un colpo di clock non appena il contatore raggiunge il valore 2603;

- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

1.1.4 Control Unit



La Control Unit permette di controllare il Data Path per eseguire l'algoritmo richiesto. Essa presenta i seguenti segnali di I/O tutti attivi alti:

- TX_REQ: Segnale per richiedere al trasmettitore l'invio di un dato;

- TC_10: Segnale di Terminal Count proveniente dal contatore modulo 10;

- TC_2604: Segnale di Terminal Count proveniente dal contatore modulo 2604;

- RESET: Segnale di reset della macchina a stati che permette di portarla allo stato IDLE;

- RESET_REGISTER: Segnale per azzerare il contenuto del registro REGISTER;

- SE: Segnale di Shift per il registro REGISTER;

- LE: Segnale di caricamento per il registro REGISTER;

- RESET_2604: Segnale di reset per il contatore modulo 2604;

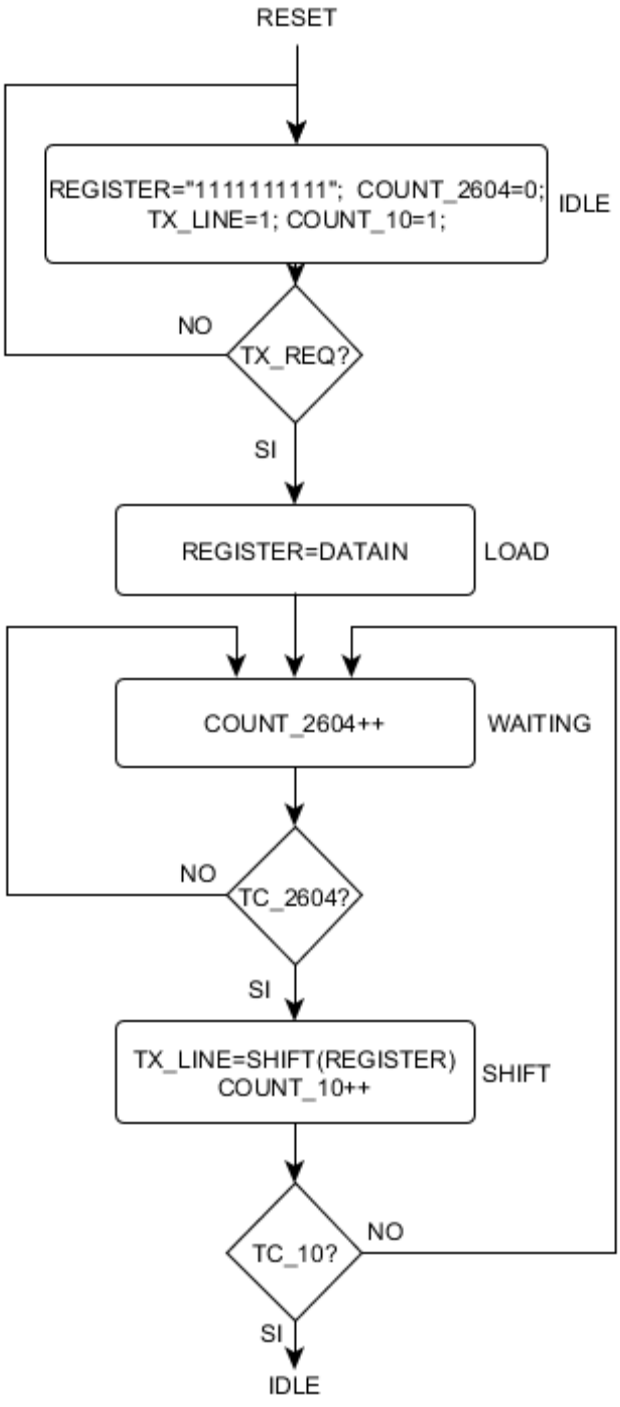
- RESET_10: Segnale di reset per il contatore modulo 10;

- TX_DONE: Segnale di uscita che viene portato a livello logico '0' durante la trasmissione;

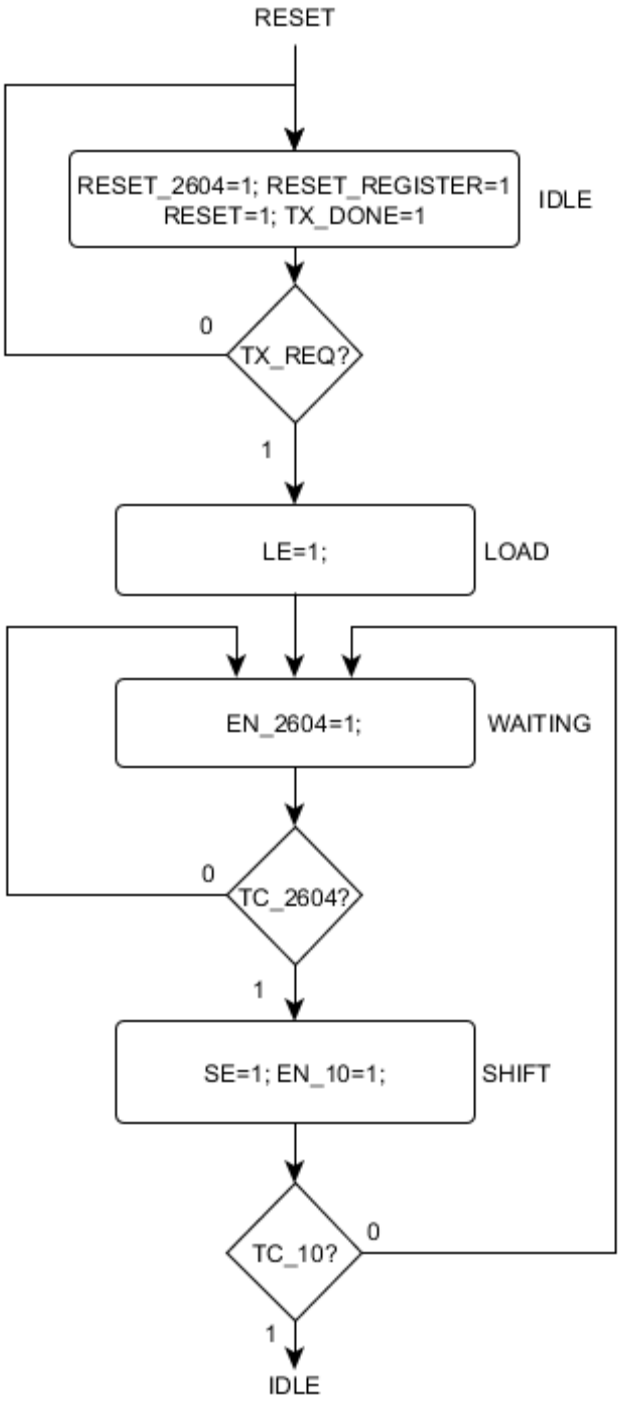
- CLOCK_50: Ingresso per il segnale di temporizzazione del sistema con frequenza 50 MHz;

- **1.1.5 Asm Chart**

Asm Data



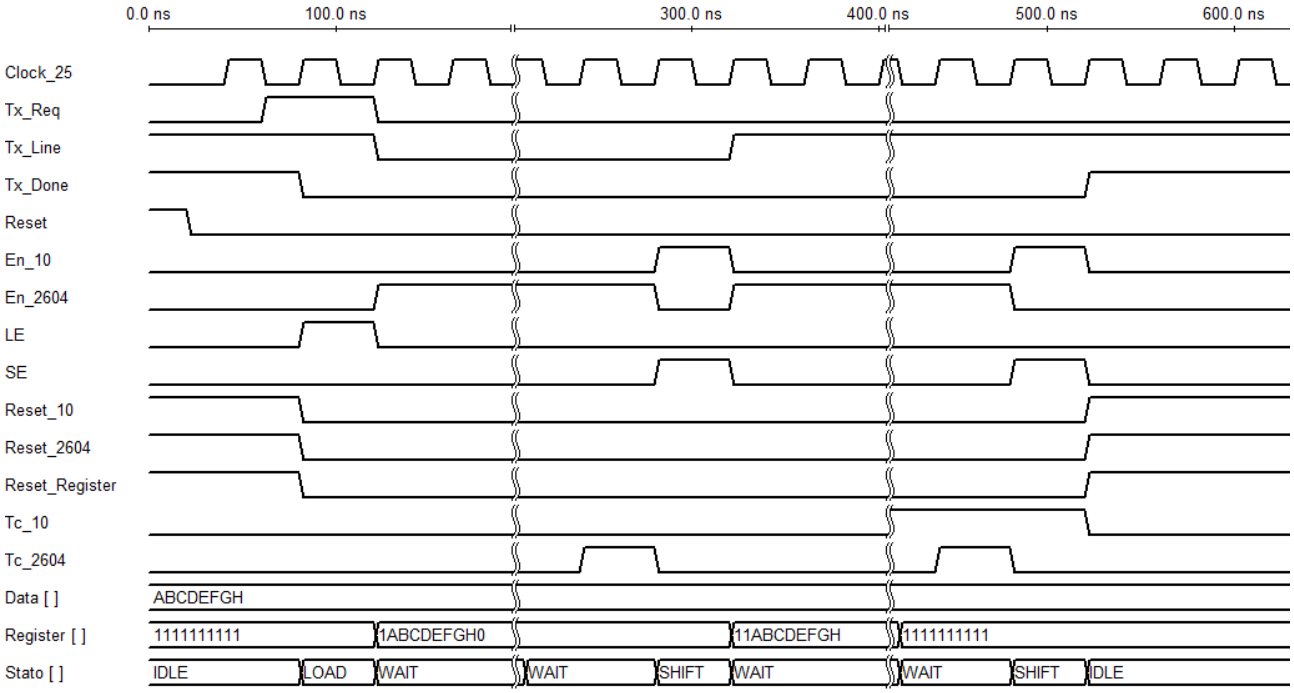
Asm Control



Nota: nei diagrammi mostrati sono considerati a zero tutti i segnali non indicati all'interno degli stati. Sono stati omessi per rendere più semplice e compatta la rappresentazione dell'algoritmo.

1.1.6 Timing

Di seguito sono mostrate tre fasi importanti del funzionamento del trasmettitore UART. Nei primi tre stati viene effettuato il Reset, il caricamento del dato da inviare (in seguito alla richiesta Tx_Req) e l'inizio del bit di Start sulla linea seriale. Successivamente viene mostrato l'invio del primo bit del dato e l'invio del bit di Stop nonché il ritorno allo stato di IDLE. Lo stato di WAIT comprende 2604 colpi di clock, utili per garantire la corretta velocità di trasmissione di 9600 bps.



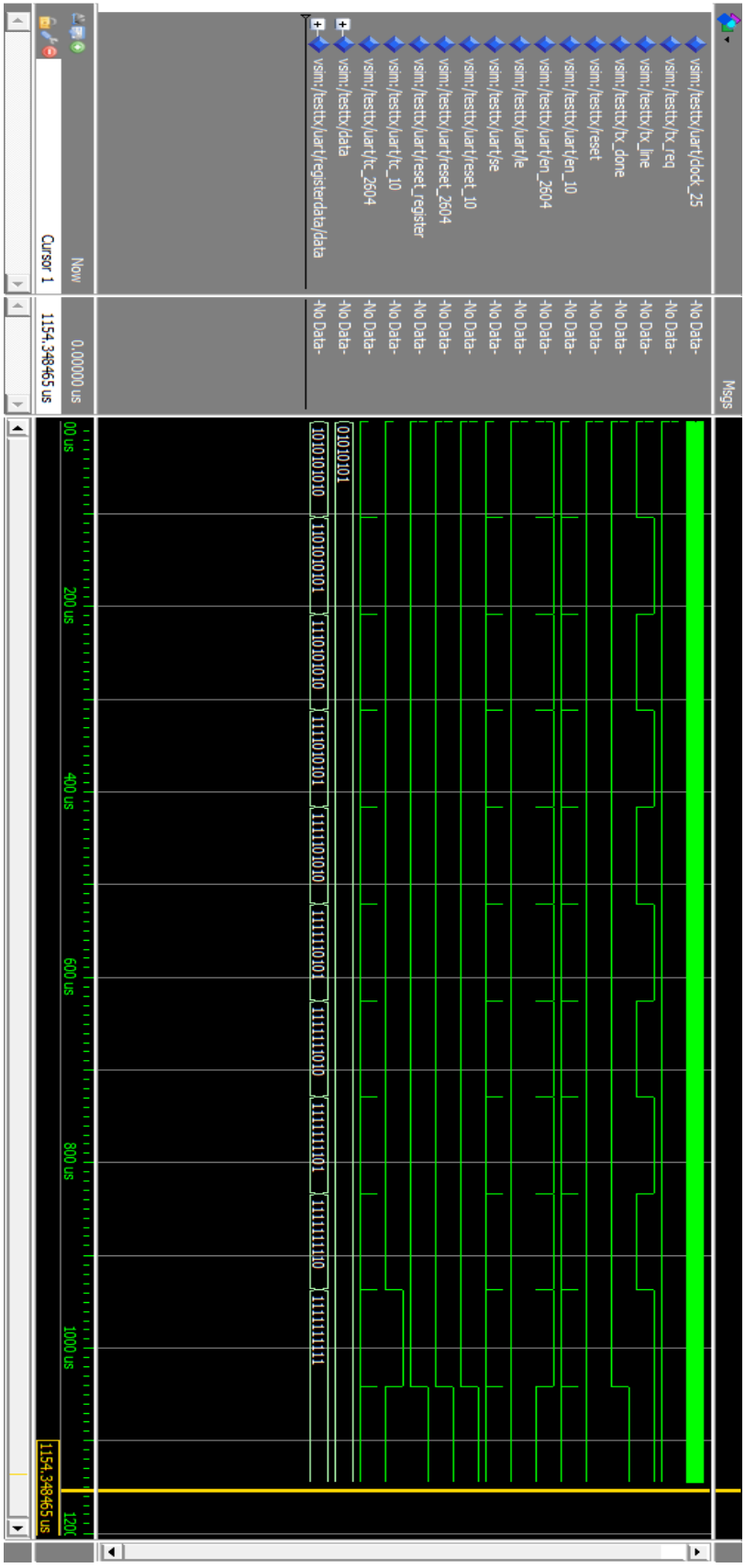
1.1.7 Testing

Di seguito è mostrato il codice VHDL della simulazione e il timing ottenuto.

Codice VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity testtx is
end entity;

architecture test of testtx is
component UARTTX is
PORT(Data_IN: IN std_logic_vector(7 downto 0);
   Tx_Req,Clock,Reset : IN std_logic;
    Tx_Done: OUT std_logic;
    Tx_Line: OUT std_logic;
    STATO: OUT std_logic_vector(1 downto 0));
    end component;


signal Data: std_logic_vector(7 downto 0);
signal Tx_Req,Clock,Reset : std_logic;
signal Tx_Done: std_logic;
signal Tx_Line: std_logic;
signal STATO: std_logic_vector(1 downto 0);

BEGIN

PROCESS
BEGIN
Reset<='1';
Data<="01010101";
Tx_req<='0';
wait for 30 ns;
Reset<='0';
wait for 40 ns;
Tx_req<='1';
wait for 40 ns;
Tx_req<='0';
wait;
end process;
PROCESS
BEGIN
clock<='1';
wait for 10 ns;
clock<='0';
wait for 10 ns;
end process;
UART: UARTTX PORT MAP ( Data,Tx_Req,Clock,Reset,Tx_Done,Tx_Line,STATO);
end architecture;
```

Timing



Come si può notare il timing ottenuto rispetta le specifiche di progetto.

1.2 Ricevitore RS232

Il ricevitore RS232 provvede alla gestione della ricezione seriale dei dati. Il dato ricevuto è memorizzato in un registro. Il ricevitore rimane in attesa del bit di START proveniente dal trasmettitore e memorizza in seguito i bit del dato fino al bit di STOP. In caso di errore nella ricezione esso abilita una linea di uscita che segnala all'utente un possibile malfunzionamento.

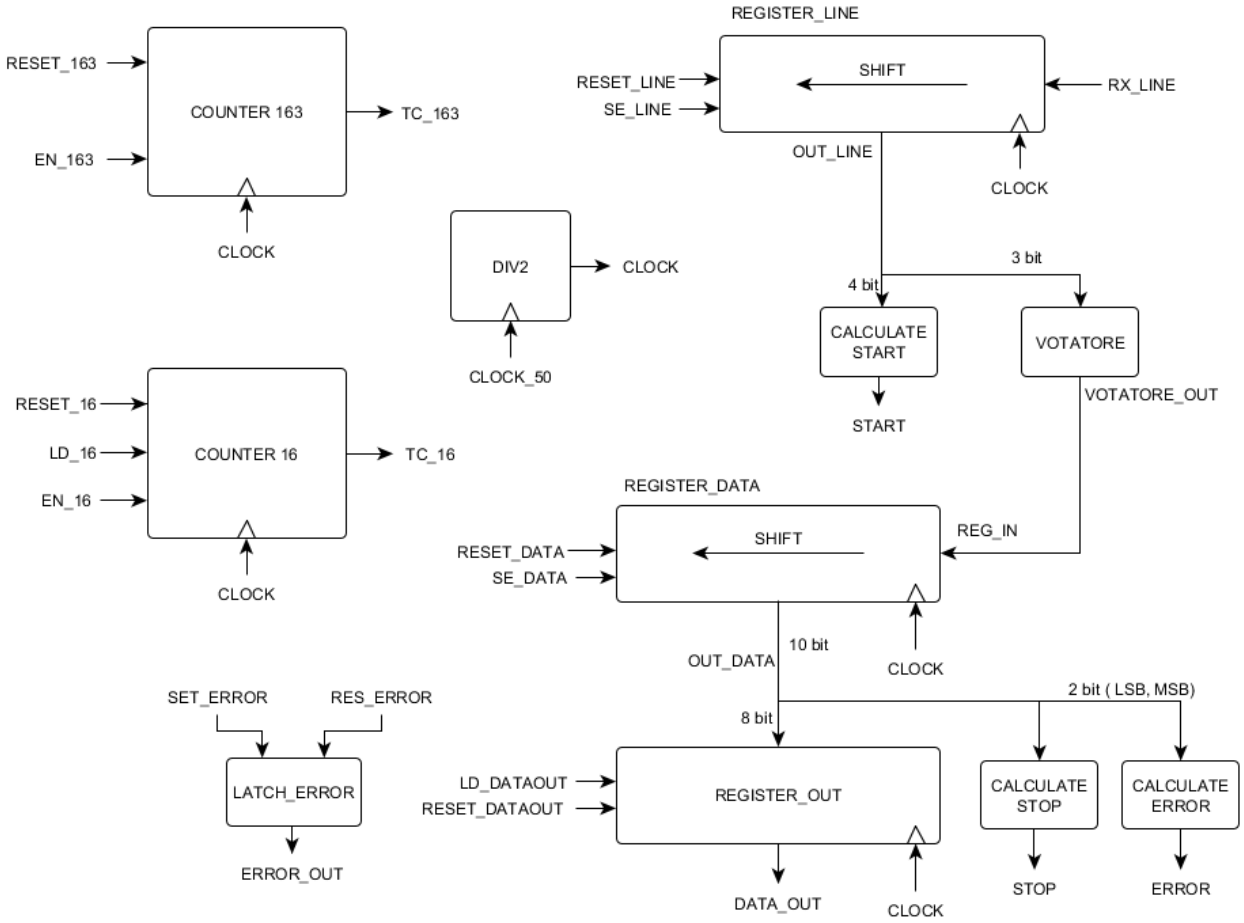
1.2.1 Specifiche

La linea seriale in assenza di trasmissione è a livello logico '1'. Il ricevitore campiona con una frequenza pari sedici volte quella del trasmettitore. Nel nostro caso essa corrisponde a 9600 x 16 = 153600 Hz. Dopo ogni istante di campionamento il ricevitore memorizza il valore campionato in un registro di parallelismo 4 bit. Lo START viene rilevato nel momento in cui avviene una transizione da livello logico 1 a livello logico 0 sulla linea seriale. Successivamente, il ricevitore si posiziona a campionare i bit del dato a metà della loro durata. Questo assicura la memorizzazione di un dato stabile. La ricezione termina con il bit di STOP. Anche in questo caso la base tempi utilizzata è un clock a 25 MHz ottenuta con un divisore per 2 dal clock di sistema di 50 MHz. È pertanto necessario utilizzare un contatore per una frequenza di campionamento di 153600 Hz. Di seguito è presentato lo pseudo-codice dell'algoritmo per realizzare la macchina a stati del ricevitore.

1.2.2 Pseudo Codice

```
REGISTER_LINE="0000";
COUNT_163=0;
COUNT_16=11;
TX_DONE=1;
while(START!=1){
for(COUNT_163=0;COUNT_163<=162;COUNT_163++);
REGISTER_LINE=SHIFT(RX_LINE);
CALCULATE_START;
}
TX_DONE=0;
for(COUNT_16<=15;COUNT_16++){
for(COUNT_163=0;COUNT_163<=162;COUNT_163++);
REGISTER_LINE=SHIFT(RX_LINE);
}
REGISTER_DATA=SHIFT(VOTATORE_OUT);
while(STOP!=1 || ERROR!=1){
for(COUNT_16=0;COUNT_16<=15;COUNT_16++){
for(COUNT_163=0;COUNT_163<=162;COUNT_163++);
REGISTER_LINE=SHIFT(RX_LINE);
}
REGISTER_DATA=SHIFT(VOTATORE_OUT);
CALCULATE_STOP AND ERROR;
}
TX_DONE=1;
```

1.2.3 Data Path



- **Register_LINE**: registro di ingresso del ricevitore, esso permette di campionare i dati sulla linea seriale;

- **Calculate_START**: blocco che permette attraverso un operazione combinatoria di rilevare la presenta del bit di START all'interno del registro Register_LINE;

- **Votatore**: blocco che permette di stabilire se il dato contenuto all'interno del registro Register_LINE è un '1' o uno '0' logico;

- **Register_DATA**: registro in cui vengono memorizzati i bit del dato ricevuto compresi i bit di START e di STOP;

- **Calculate_STOP**: blocco combinatorio che permette di stabilire se è stato ricevuto il bit di STOP;

- **Calculate_ERROR**: blocco combinatorio che permette di stabilire se è vi è un errore durante la ricezione;

- **Latch_ERROR**: latch SR che permette di asserire la linea di uscita Error_Out in corrispondenza di un errore di ricezione;

- **Register_OUT**: registro in cui viene memorizzato il byte relativo al dato ricevuto asserendo la linea TX_Done;

- **Counter_163**: contatore modulo 163 per generare la frequenza di ricezione richiesta;

- **Counter_16**: contatore modulo 16 utilizzato per campionare i dati a metà della loro durata temporale;

- **DIV/2**: blocco che permette di ottenere da clock di sistema un segnale di temporizzazione a 25 MHz attraverso un divisore per 2;

Calculate Start

Il blocco **Calculate Start** permette attraverso una rete combinatoria di determinare la ricezione del bit di START analizzando il contenuto del registro **Register_LINE**. In particolare esso realizzare la funzione logica:

Start<= (NOT **Register_LINE**(3)) AND (NOT **Register_LINE** (2)) AND **Register_LINE** (1) AND **Register_LINE** (0);

Viene così rilevata una transizione da ‘1 a ‘0’ logico della linea seriale.

Votatore

Il blocco **Votatore** è una rete combinatoria che analizza i 3 bit più significativi del registro **Register_LINE** e fornisce il valore da caricare nel registro **Register_DATA**. Esso permette quindi di stabilire attraverso una semplice valutazione di maggioranza il bit ricevuto.

Calculate Stop

Il blocco **Calculate Stop** permette di verificare quando è stato ricevuto un byte correttamente ossia quando il numero di bit ricevuti è pari a 10. Considerando il registro **Register_DATA** inizialmente a “1111111111”, se viene ricevuto un dato valido, il contenuto atteso è **Register_DATA** = “1XXXXXXXX0”. Questo significa che è stato ricevuto un bit di START, un dato ed un corretto bit di STOP. Pertanto Stop = (**Register_DATA**(9)) AND (NOT (**Register_DATA**(0))).

Calculate Error

Il blocco **Calculate Error** permette di verificare se è presente un errore di ricezione. Considerando il registro **Register_DATA** inizialmente a “1111111111” se accade che, in seguito alla ricezione di un dato, il contenuto diventa “0XXXXXXXX0” significa che la trasmissione è terminata con un bit STOP non corretto. Pertanto:

Error = NOT(**Register_DATA**(9)) AND (NOT (**Register_DATA**(0))).

Register_OUT

Il registro **Register_OUT** permette, terminata la ricezione dei dati con il bit di STOP, di presentare sulla porta di uscita **DATA_OUT** il dato ricevuto. Esso ha un parallelismo a 8 bit. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RES_OUT: segnale di reset che in seguito alla sua asserzione porta il contenuto del registro REGISTER_OUT a “00000000”;
- LD_OUT: segnale di Load Enable. Esso permette di abilitare il caricamento del dato presente in ingresso al **Register_OUT**;
- DATA_OUT: linea di uscita a 8 bit che permette di fornire all’utente il dato ricevuto;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Register_DATA

Il registro **Register_DATA** è uno Shift Register con caricamento seriale e parallelismo a 10 bit. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RES_DATA: segnale di reset che in seguito alla sua asserzione porta il contenuto del registro **Register_DATA** a “1111111111”;
- SE_DATA: segnale che permette di abilitare lo shift sincrono del dato presente all’ingresso REG_IN;
- REG_IN: linea di ingresso dello Shift Register;
- OUT_DATA: porta di uscita a 10 bit che permette di leggere il dato contenuto nel registro REGISTER_DATA;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Register_LINE

Il registro **Register_LINE** è uno Shift-Register con caricamento seriale e parallelismo a 4 bit. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RES_LINE: segnale di reset che in seguito alla sua asserzione porta il contenuto del registro **Register_LINE** a "0000";
- SE_LINE: segnale che permette di abilitare lo shift sincrono del dato presente all'interno del registro **Register_LINE** sostituendo l'MSB con il valore presente all'ingresso RX_LINE;
- RX_LINE: linea di ingresso per il collegamento con il trasmettitore;
- OUT_LINE: linea di uscita che permette di leggere il dato contenuto nel registro **Register_LINE**;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Counter_16

Il contatore **Counter_16** è un contatore modulo 16. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RESET_16: segnale di reset che permette di azzerare il contatore portando il segnale TC_16 al valore logico '0';
- EN_16: segnale di ingresso sincrono di abilitazione del contatore;
- TC_16: segnale di uscita Terminal Count. Esso rimane attivo per un colpo di clock non appena il contatore raggiunge il valore 15;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Counter_163

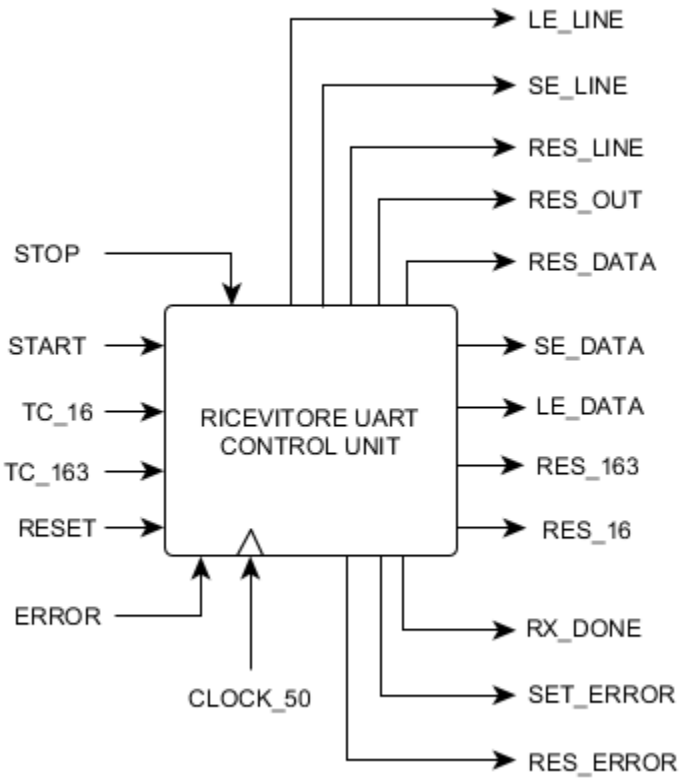
Il contatore **Counter_163** è un contatore modulo 163. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RESET_163: segnale di reset che permette di azzerare il contatore portando il segnale TC_163 al valore logico '0';
- EN_163: segnale di ingresso sincrono di abilitazione del contatore;

- TC_163: segnale di uscita Terminal Count. Esso rimane attivo per un colpo di clock non appena il contatore raggiunge il valore 162;

- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

1.2.4 Control Unit



La Control Unit permette di controllare il Data Path per eseguire l'algoritmo richiesto. Essa presenta i seguenti segnali di I/O tutti attivi alti:

- STOP: segnale ricevuto dal Data Path alla ricezione del bit di STOP;

- TC_16: segnale di Terminal Count proveniente dal contatore modulo 16;

- TC_163: segnale di Terminal Count proveniente dal contatore modulo 163;

- RESET: segnale di reset della macchina a stati che permette di riportarla allo stato IDLE;

- ERROR: segnale ricevuto dal Data Path in caso di mancata ricezione del bit di STOP;

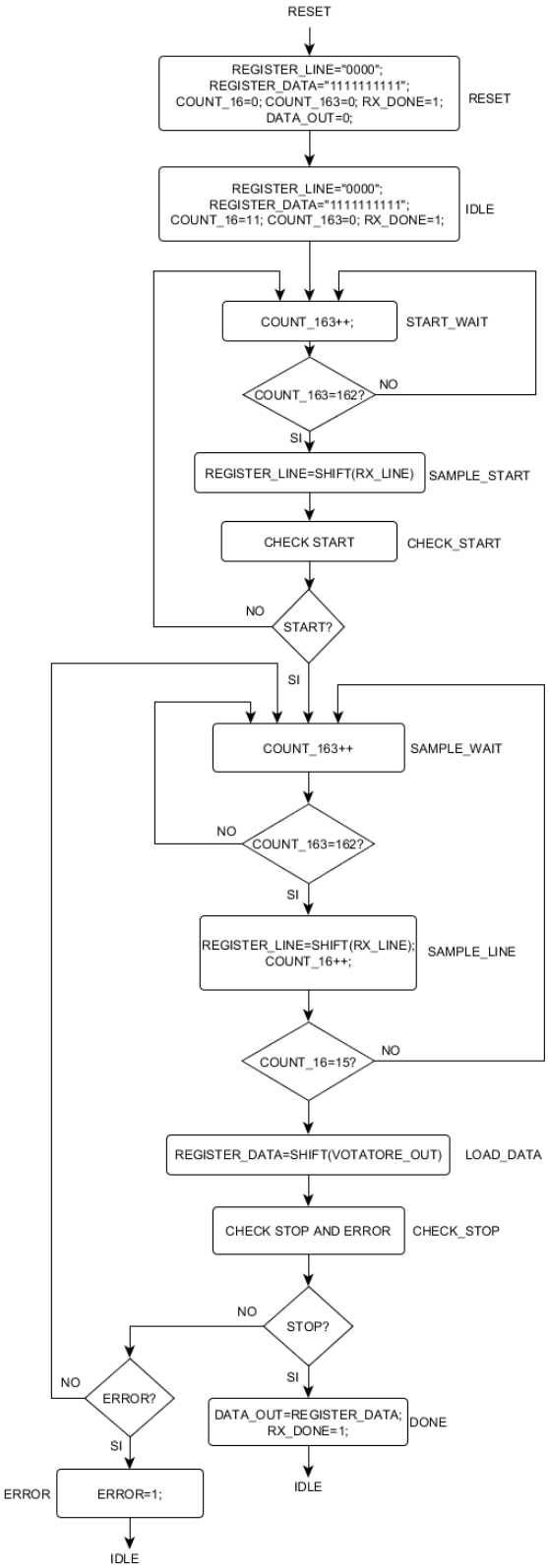
- LE_LINE: segnale di abilitazione per lo Shift del registro **Register_LINE**;

- SE_LINE: segnale di caricamento per il registro **Register_LINE**;

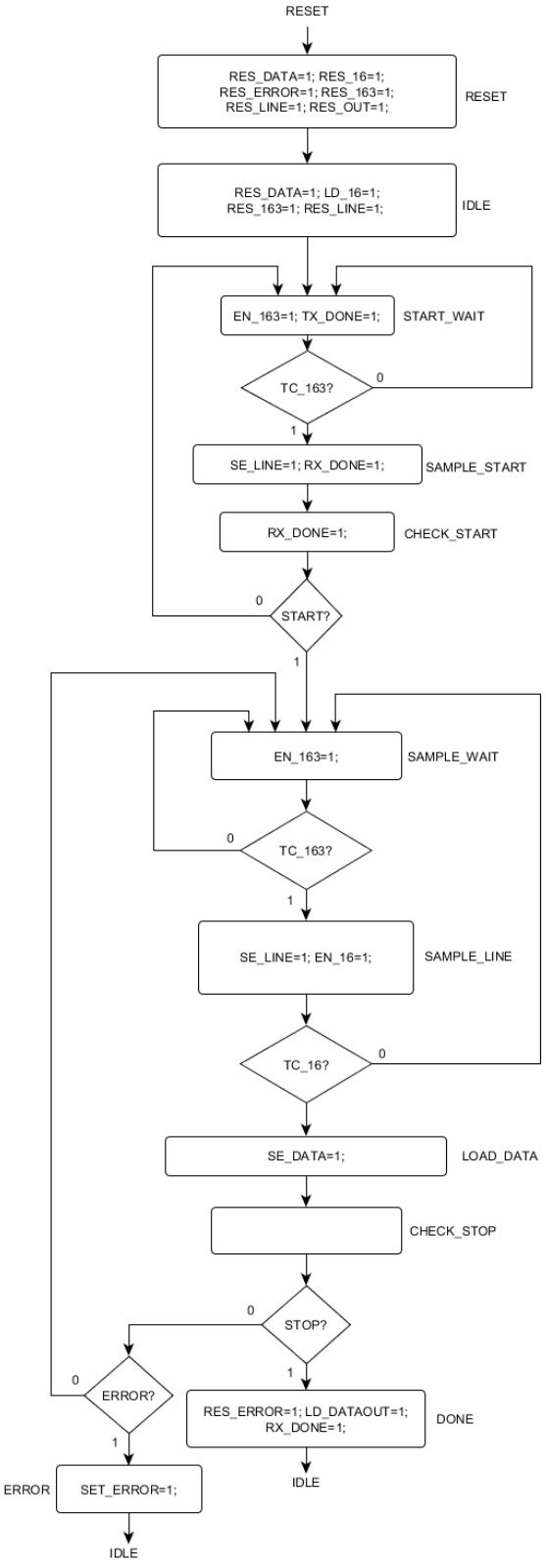
- RES_LINE: segnale per resettare il contenuto del registro **Register_LINE**;
- RES_OUT: segnale per resettare il contenuto del registro **Register_OUT**;
- RES_DATA: segnale per resettare il contenuto del registro **Register_DATA**;
- SE_DATA: segnale di abilitazione per lo Shift del registro **Register_DATA**;
- LE_DATA: segnale di caricamento per il registro **Register_DATA**;
- RES_163: segnale di reset per il contatore modulo 163;
- RES_16: segnale di reset per il contatore modulo 16;
- RX_DONE: Segnale di uscita che viene forzato a livello logico 1 quando è disponibile un dato;
- SET_ERROR: segnale per asserire il bit di ERROR;
- RES_ERROR: segnale per azzerare il bit di ERROR;
- CLOCK: Ingresso per il segnale di temporizzazione del sistema con frequenza 25 MHz;

1.2.5 Asm Chart

Asm Data

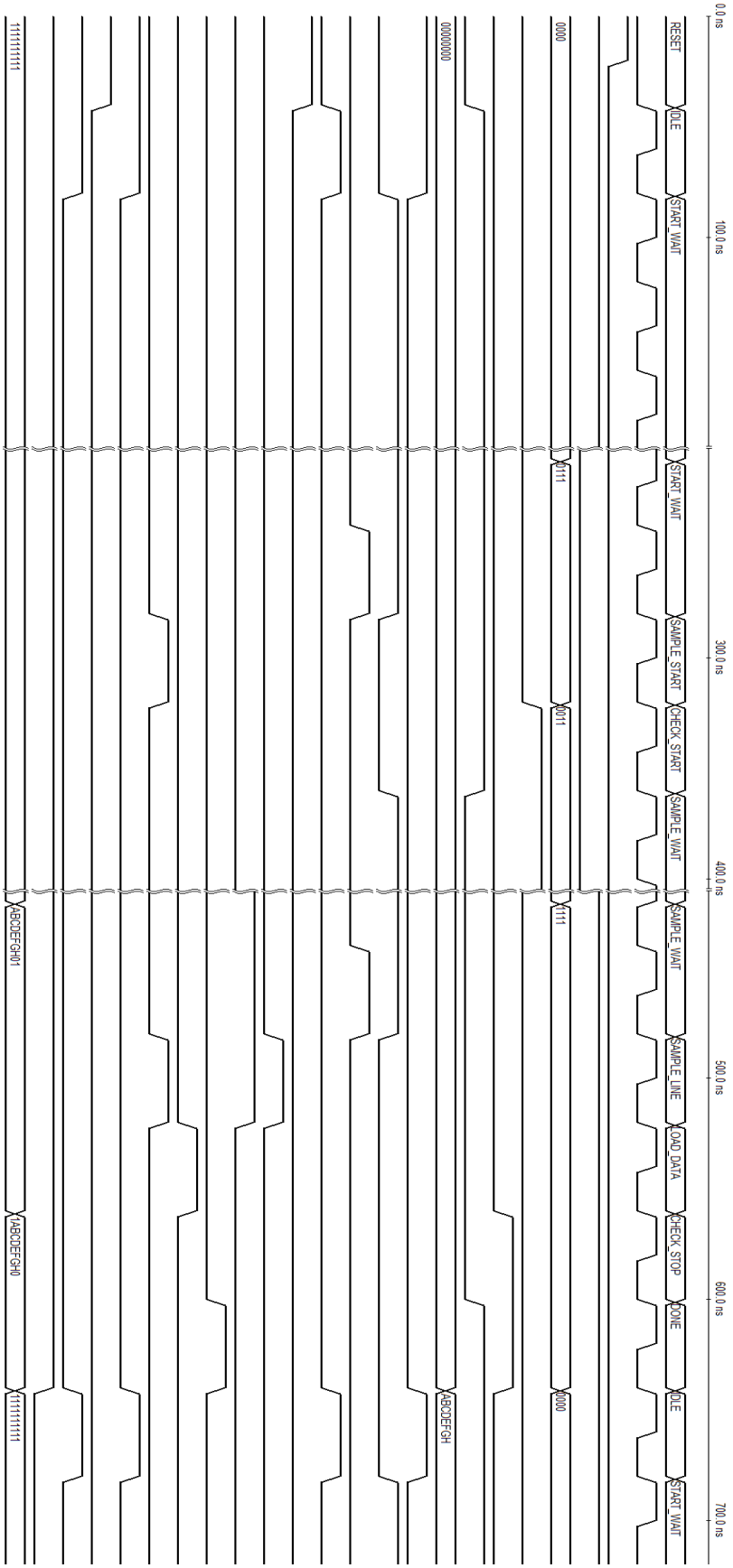


Asm Control



Nota: nei diagrammi mostrati sono considerati a zero tutti i segnali non indicati all'interno degli stati. Sono stati omessi per rendere più semplice e compatta la rappresentazione dell'algoritmo.

1.2.6 Timing


0.0 ns
100.0 ns
300.0 ns
400.0 ns
500.0 ns
600.0 ns
700.0 ns
Stato [ ]
Clock_25
Reset
Rx_Line
Line_Data [ ]
Start
Stop
RX_Done
Data_OUT [ ]
Res_163
En_163
Tc_163
LD_16
Res_16
En_16
Tc_16
LD_Data
SE_Rx
SE_Line
Res_Rx
Res_Data
Res_Line
Votatore_OUT
Data_Rx [ ]
RESET
IDLE
START_WAIT
SAMPLE_START
CHECK_START
SAMPLE_WAIT
SAMPLE_LINE
LOAD_DATA
CHECK_STOP
DONE
0000
0111
0011
00000000
ABCDEFGH
ABCDEFGH01
ABCDEFGH0
11111111111

Nell'immagine soprastante sono mostrati gli stati principali del ricevitore. Come si può osservare è presenta la fase di RESET, di IDLE, il campionamento dello START, la sua rilevazione e l'inizio della ricezione fino al bit di STOP.

1.2.7 Testing

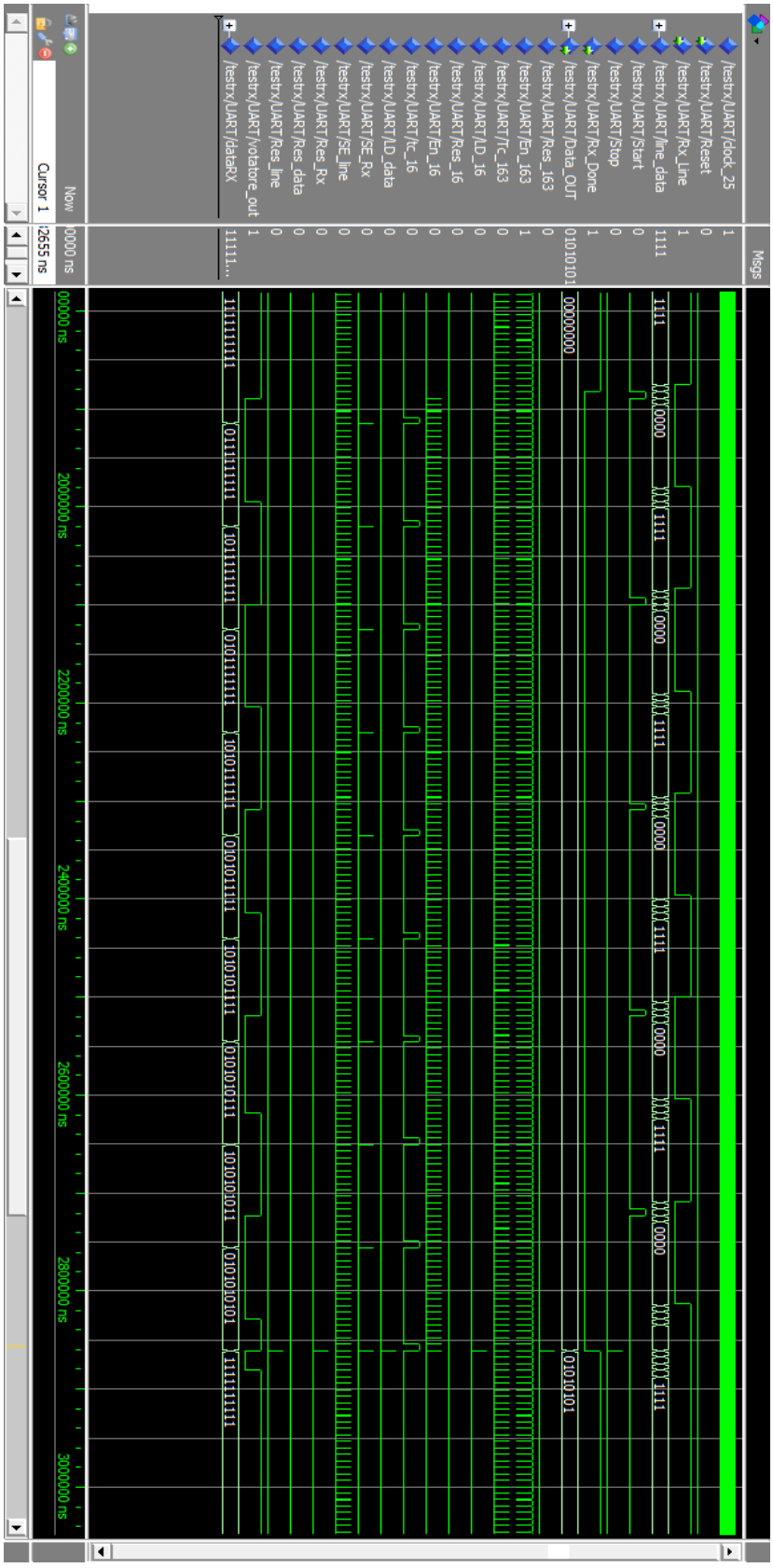
Di seguito è mostrato il timing ottenuto simulando il ricevitore con il software ModelSim. È inoltre presentato il codice VHDL del TestBench.

Codice VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity testRx is
end entity;

architecture test of testRx is
component UARTRX is
PORT(Data_OUT: OUT std_logic_vector(7 downto 0);
    Clock,Reset : IN std_logic;
    Rx_Done: OUT std_logic;
    Rx_Line: IN std_logic;
    Error_out: OUT std_logic;
    STATO: OUT std_logic_vector(3 downto 0));
end component;
signal Clock,Reset : std_logic;
signal Rx_Done: std_logic;
signal Tx_Line,Error_out: std_logic;
signal Data_OUT: std_logic_vector( 7 downto 0);
signal STATO: std_logic_vector(3 downto 0);
BEGIN

PROCESS
BEGIN
Reset<='1';
Tx_LINE<='1';
wait for 30 ns;
Reset<='0';
wait for 40 ns;
wait for 416640 ns;  --MANDiAMO 4 dati 1
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
```

```
wait for 104160 ns;
Tx_line<='0';      --Stop
wait for 104160 ns;
Tx_line<='1';      --Stop
wait for 416640 ns;  --MANDiAMO 4 dati 1
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit zero
wait for 104160 ns;
wait ;
end process;
PROCESS
BEGIN
clock<='1';
wait for 10 ns;
clock<='0';
wait for 10 ns;
end process;

UART: UARTRX PORT MAP( Data_out,clock,Reset,RX_done,Tx_line,Error_out,STATO);

end architecture;
```

Timing

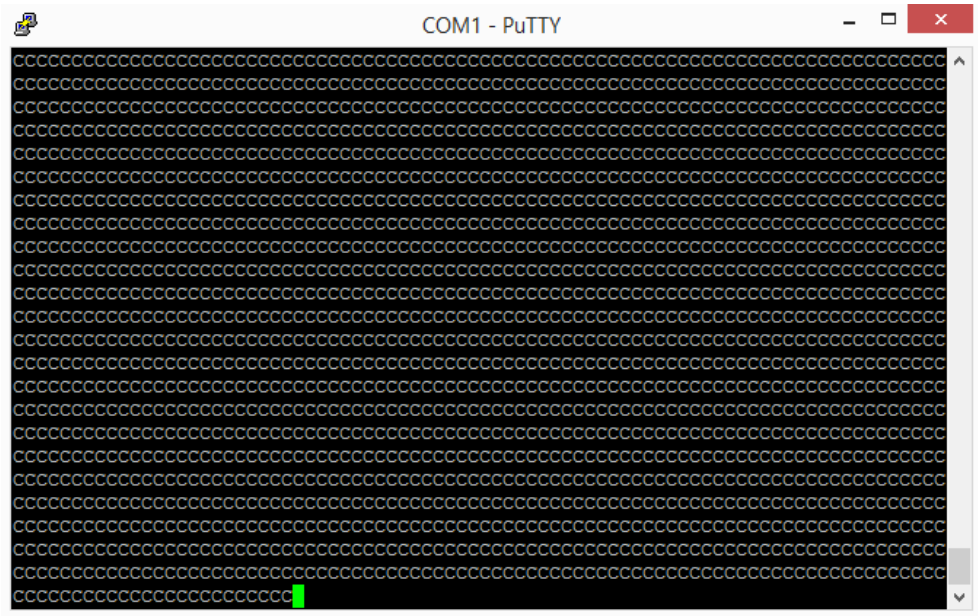


Nell'immagine soprastante è mostrato un ciclo completo di ricezione del dato "01010101" inviato sulla linea seriale. Come si può notare infatti al termine della trasmissione esso viene presentato in uscita su Data_OUT.

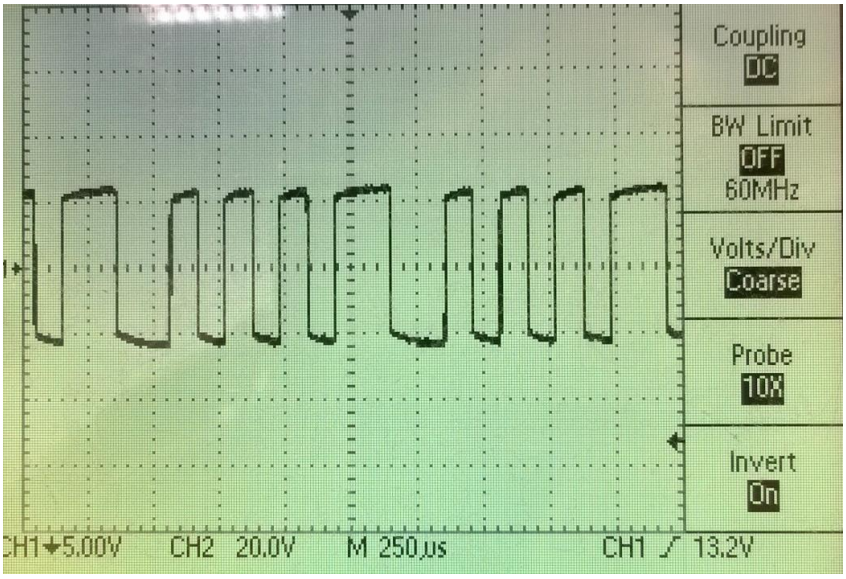
1.3 Test in Laboratorio

Dopo il progetto del trasmettitore e del ricevitore UART sono state verificare le loro funzionalità in laboratorio con l'ausilio di un PC Desktop e della scheda DE2 Board.

1.3.1 Trasmettitore

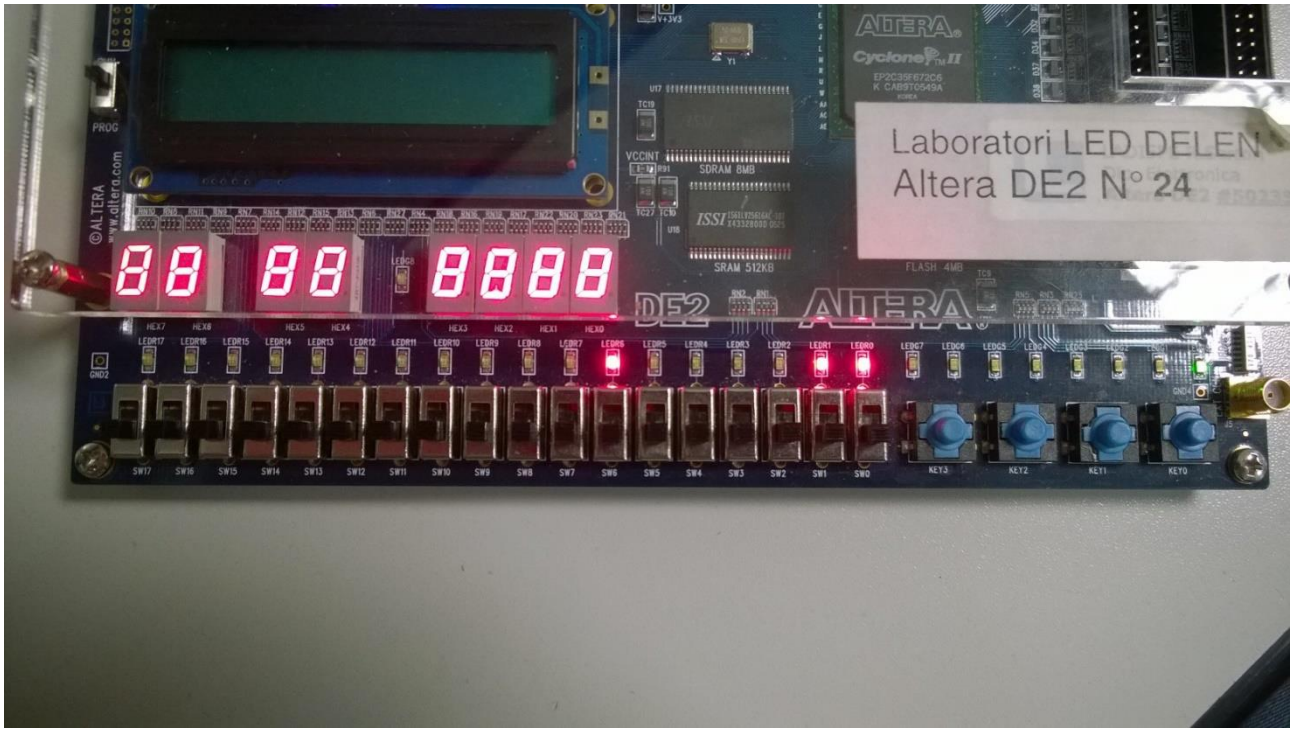


Sfruttando gli interruttori presenti sulla scheda di sviluppo è stato inviato il dato ASCII 'C'. Come si può notare, grazie all'ausilio di una console UART sul Computer è verificato il corretto invio del carattere. Di seguito è mostrata la linea Seriale TX, visualizzata con l'oscilloscopio.



1.3.2 Ricevitore

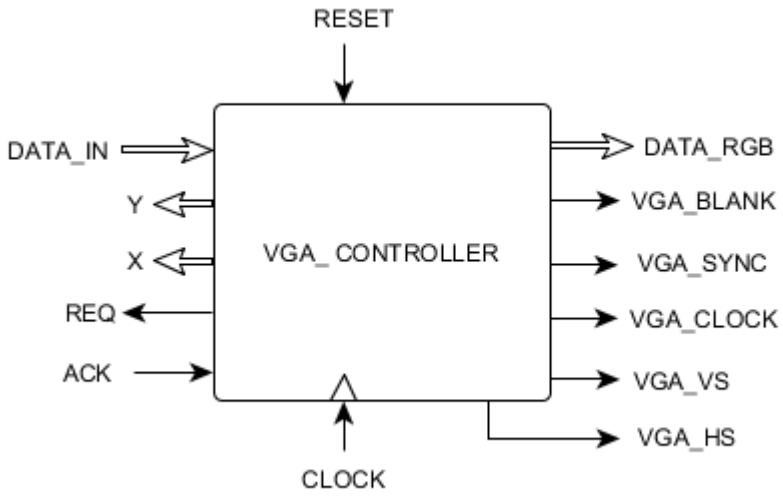
Per verificare il corretto funzionamento del ricevitore UART è stato inviato tramite il PC il carattere C. Il dato ricevuto è visualizzato tramite i led rossi presenti sulla scheda DE2 Board.



Come si può notare il dato binario rappresentato nei primi otto led corrisponde al carattere ASCII ‘C’ -> 1000011. Il Led acceso indica il valore binario 1.

2.0 Progetto VGA Controller

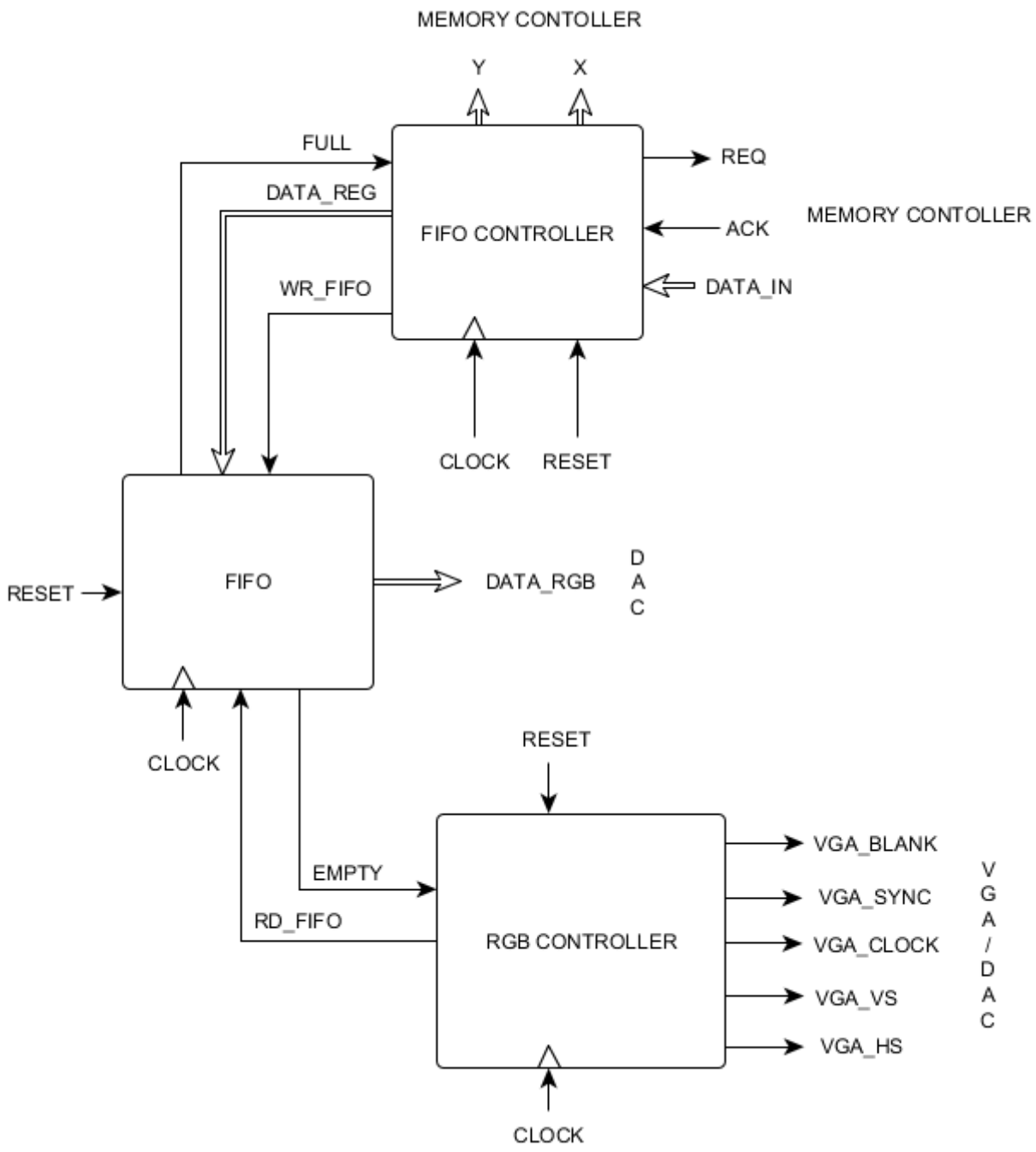
Il blocco VGA Controller permette di rappresentare un frame di dati ricevuti dall'esterno sul monitor. In particolare, i dati sono contenuti in una memoria SRAM esterna. Di seguito è rappresentato l'insieme delle porte I/O e dei segnali di controllo:



- DATA_IN: Ingresso dei dati provenienti dall'esterno;
- X, Y: segnali inviati dal VGA Controller per richiedere all'esterno un pixel in una data coordinata dello schermo;
- REQ, ACK: segnali di handshake che permettono al VGA Controller di scambiare i dati con l'esterno;
- RESET: segnale per portato allo stato di RESET il VGA Controller;
- CLOCK: ingresso del segnale di temporizzazione;
- DATA_RGB: uscita dei dati inviati al convertitore DAC per generare i segnale R, B, G;
- VGA_HS, VGA_VS: insieme dei segnali di sincronismo del protocollo VGA;
- VGA_SYNC, VGA_BLANK, VGA_CLOCK: insieme dei segnali di controllo per il convertitore DAC;

Le richieste del VGA Controller vengono elaborate dal Memory Controller.

Il VGA Controller è costituito internamente da tre blocchi:

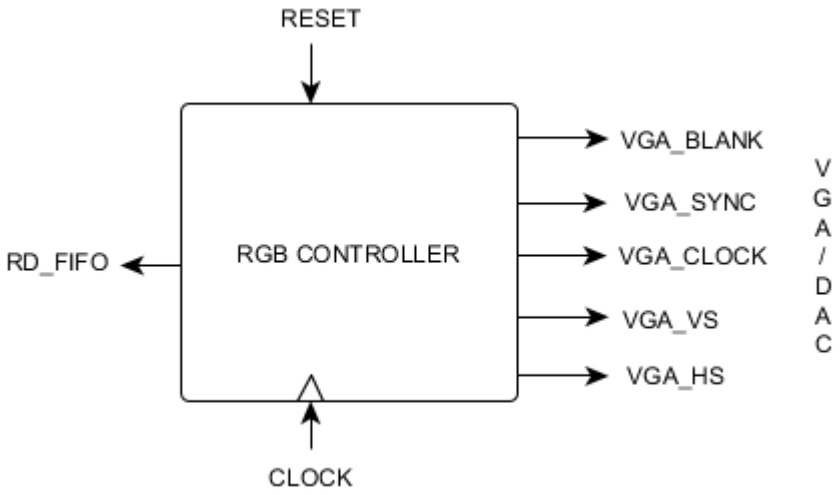


- RGB Controller: blocco che fornisce i segnali di sincronismo per il protocollo VGA e comanda il convertitore DAC;

- FIFO: memoria utilizzata dal VGA Controller per precaricare all'interno, i dati da rappresentare sul monitor, in modo da garantirne sempre l'esistenza in caso di mancata risposta della Memory Controller;

- FIFO Controller: macchina a stati che gestisce il caricamento dei dati all'interno della FIFO;

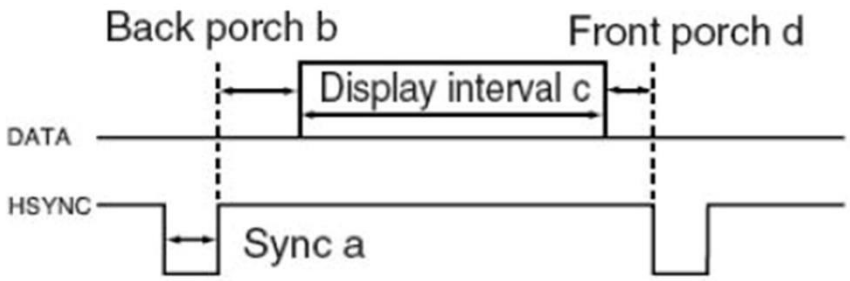
2.1 RGB Controller



2.1.1 Specifiche

Il blocco RGB Controller provvede a generare i segnali di sincronismo verticale e orizzontale per il protocollo VGA. In parallelo, viene controllato anche il convertitore DAC per ottenere i segnali R, G, B. Di seguito è mostrato il timing del protocollo VGA:

Horizontal Sync:



Nel nostro caso, utilizzando una risoluzione di 680x480 i tempi A_h, B_h, C_h, D_h valgono rispettivamente:

A_h= 3.8 µs, B_h = 1.9 µs, C_h = 25.4 µs, D_h=0.6 µs.

Vertical Sync:

Il vertical Sync presenta lo stesso Timing dell'Horizontal Sync. L'unica differenza si ha nella durata dei tempi A, B, C e D. In particolare essi sono uguali a:

A_V= 2 Linee, B_V = 33 Linee, C_h = 480 Linee, D_h=10 Linee.

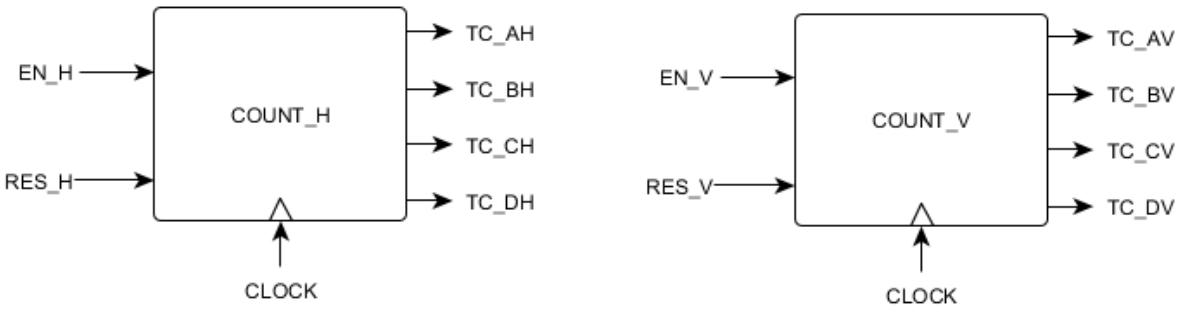
Dove ogni Linea corrisponde al tempo impiegato per la generazione di un ciclo di Horizontal Sync.

Il convertitore DAC, invece, necessita del dato digitale DATA_RGB e dei segnali VGA_BLANK, VGA_SYNC e VGA_CLOCK. Il timing è mostrato sul Datasheet del componente. Il segnale

VGA_SYNC non viene utilizzato mentre VGA_BLANK viene forzato a 1 logico solo durante il tempo C_h. Normalmente viene forzato a livello logico '0' garantendo che i dati R, G e B siano a valore 0.

2.1.2 Data Path:

Il Data Path è costituito da due contatori che permettono di creare i segnali di sincronismo verticale e orizzontale. Di seguito sono descritti in modo dettagliato.



Counter_H

Il contatore **Counter_H** è un contatore modulo 800. Esso permette di generare i tempi A_h, B_h, C_h, D_h attivando il rispettivo Terminal Count. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RESET_H: segnale di reset che permette di azzerare il contatore portando tutti i Terminal Count al valore logico '0';
- EN_H: segnale di ingresso sincrono di abilitazione del contatore;
- TC_AH: Terminal Count attivato dopo il tempo A_h;
- TC_BH: Terminal Count attivato dopo il tempo B_h;
- TC_CH: Terminal Count attivato dopo il tempo C_h;
- TC_DH: Terminal Count attivato dopo il tempo D_h;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo. È richiesto un clock a 25.175 MHz;

Counter_V

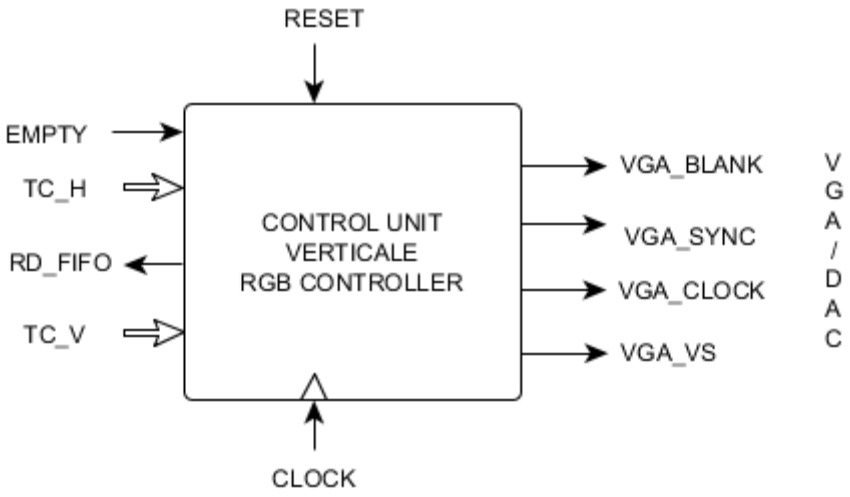
Il contatore **Counter_V** è un contatore modulo 420000. Esso permette di generare i tempi A_v, B_v, C_v, D_v attivando il rispettivo Terminal Count. Tutti i segnali di controllo sono attivi alti. Di seguito sono elencati i segnali di I/O:

- RESET_V: segnale di reset che permette di azzerare il contatore portando tutti Terminal Count al valore logico ‘0’;
- EN_V: segnale di ingresso sincrono di abilitazione del contatore;
- TC_AV: Terminal Count attivato dopo il tempo A_v;
- TC_BV: Terminal Count attivato dopo il tempo B_v;
- TC_CV: Terminal Count attivato dopo il tempo C_v;
- TC_DV: Terminal Count attivato dopo il tempo D_v;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo. È richiesto un clock a 25.175 MHz;

2.1.3 Control Unit

Internamente sono state realizzate due Control Unit differenti che condividono lo stesso Data Path. Questo permette di generare i segnali di sincronismo correttamente. La prima Control Unit è utilizzata per il sincronismo verticale e l’invio dei dati al convertitore DAC durante il tempo C_h. La seconda invece è utilizzata per generare in modo continuo il segnale di sincronismo orizzontale.

Control Unit Sincronismo Verticale

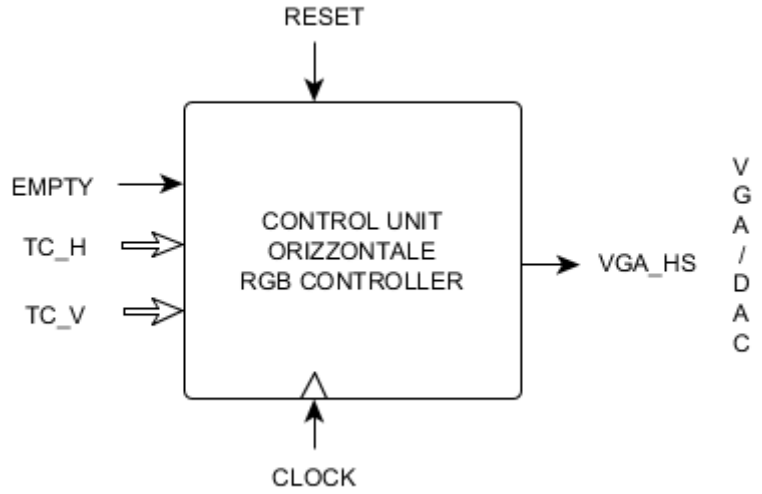


Essa presenta i seguenti segnali di I/O tutti attivi con livello logico ‘1’:

- TC_H: insieme dei Terminal Count proveniente da **Counter_H**;
- TC_V: insieme dei Terminal Count proveniente da **Counter_V**;

- EMPTY: segnale inviato dalla FIFO. Esso viene attivato quando la memoria è vuota;
- RD_FIFO: segnale di RD_FIFO attivo durante l'invio di una riga di pixel;
- VGA_BLANK, VGA_SYNC, VGA_CLOCK: segnali di controllo inviati al convertitore DAC;
- VGA_VS: segnale di sincronismo verticale dello standard VGA;
- RESET: segnale per azzerare la macchina a stati;
- CLOCK: ingresso del segnale di temporizzazione. È richiesto un clock di 25.175 MHz;

Control Unit Sincronismo Orizzontale



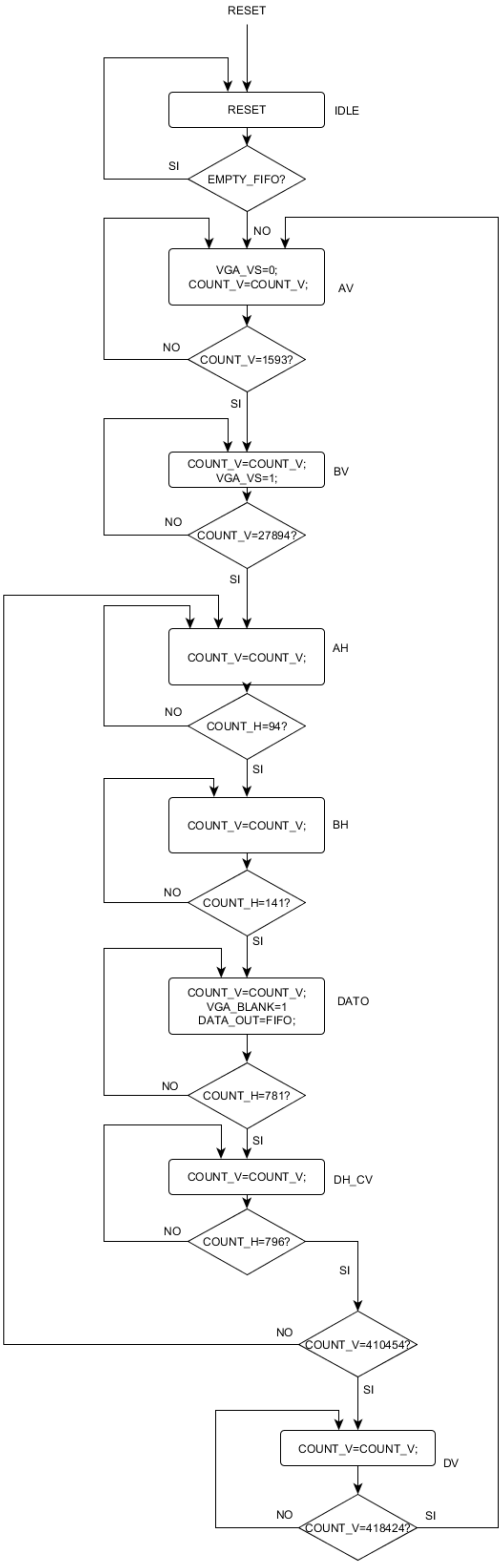
La Control Unit permette di controllare il Data Path per eseguire l'algoritmo richiesto. Essa presenta i seguenti segnali di I/O tutti attivi alti:

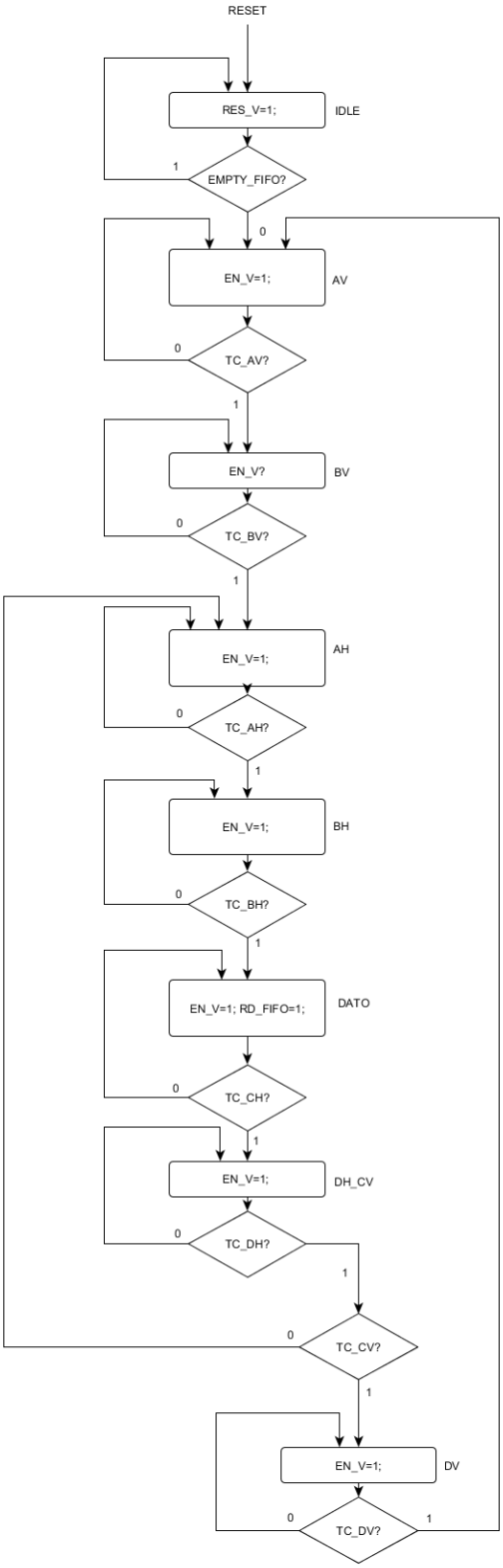
- TC_H: insieme dei Terminal Count proveniente da **Counter_H**;
- TC_V: insieme dei Terminal Count proveniente da **Counter_V**;
- EMPTY: segnale inviato dalla FIFO. Esso viene attivato quando la memoria è vuota;
- VGA_HS: segnale di sincronismo orizzontale dello standard VGA;
- RESET: segnale per azzerare la macchina a stati;
- CLOCK: ingresso del segnale di temporizzazione. È richiesto un clock di 25.175 MHz;

2.1.4 Asm Charts

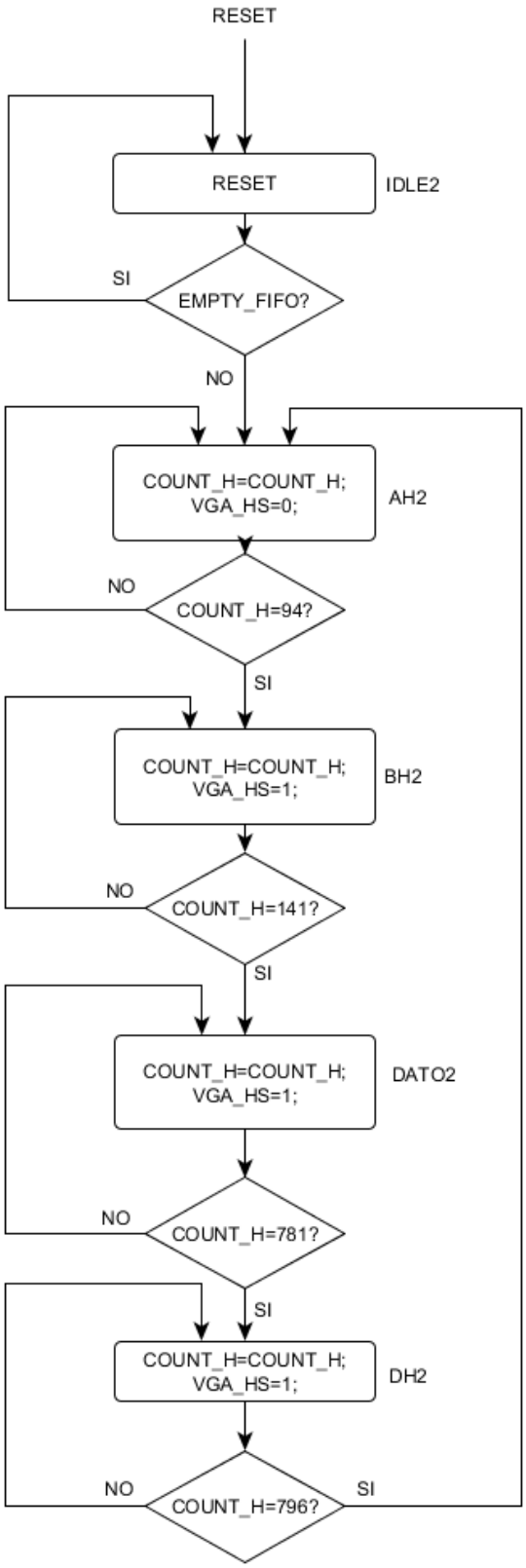
Di seguito sono mostrati i diagrammi per la realizzazione delle macchine a stati. In particolare vengono presentati gli Asm Data e Asm Control. Per semplificarne la lettura vengono indicati solo i segnali il cui valore è diverso dal Default.

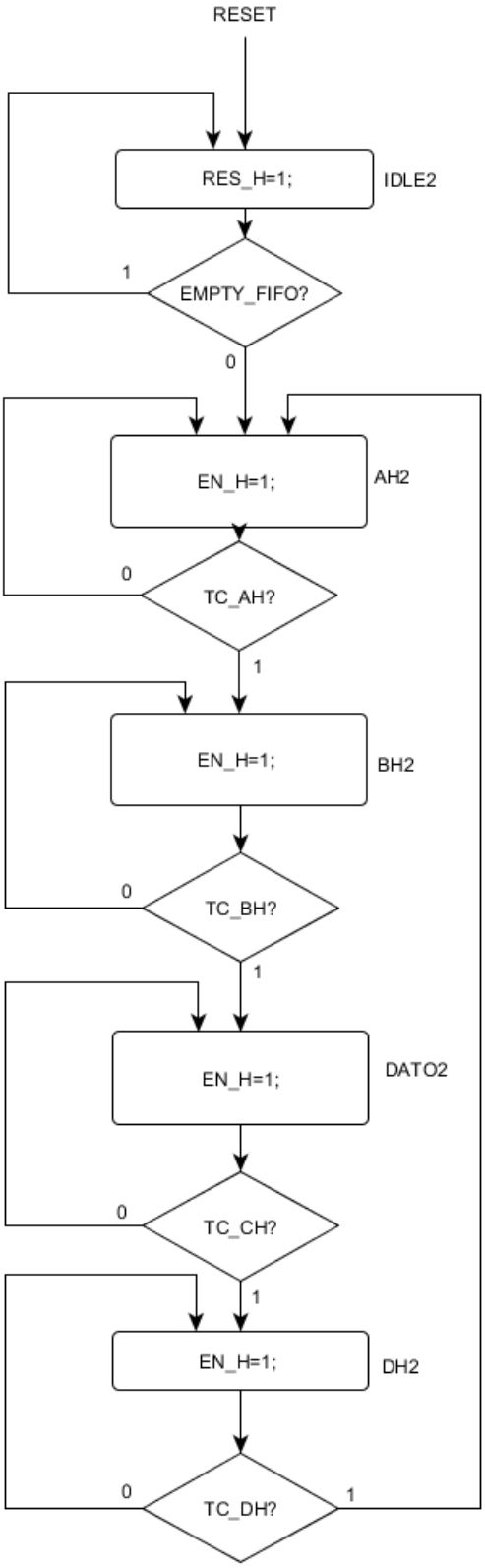
Control Unit Verticale





Control Unit Orizzontale

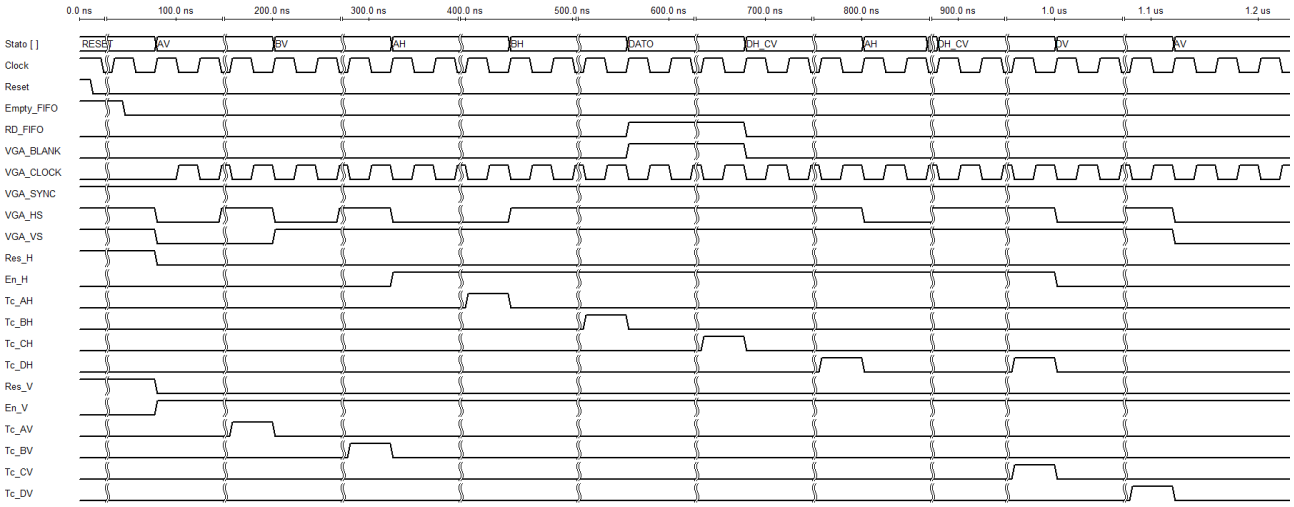




Nota: nei diagrammi mostrati sono considerati a zero tutti i segnali non indicati all'interno degli stati. Sono stati omessi per rendere più semplice e compatta la rappresentazione dell'algoritmo.

2.1.5 Timing:

Di seguito è mostrato il Timing del RGB Controller. Come si può notare, sono stati rappresentati solo gli stati principali in quanto, un ciclo completo, impiega circa 17 ms.



2.1.6 Testing:

Attraverso il software ModelSim è stato verificato il corretto funzionamento del blocco RGB Controller. Di seguito è presentato il codice VHDL della simulazione e i timing ottenuti.

Codice VHDL:

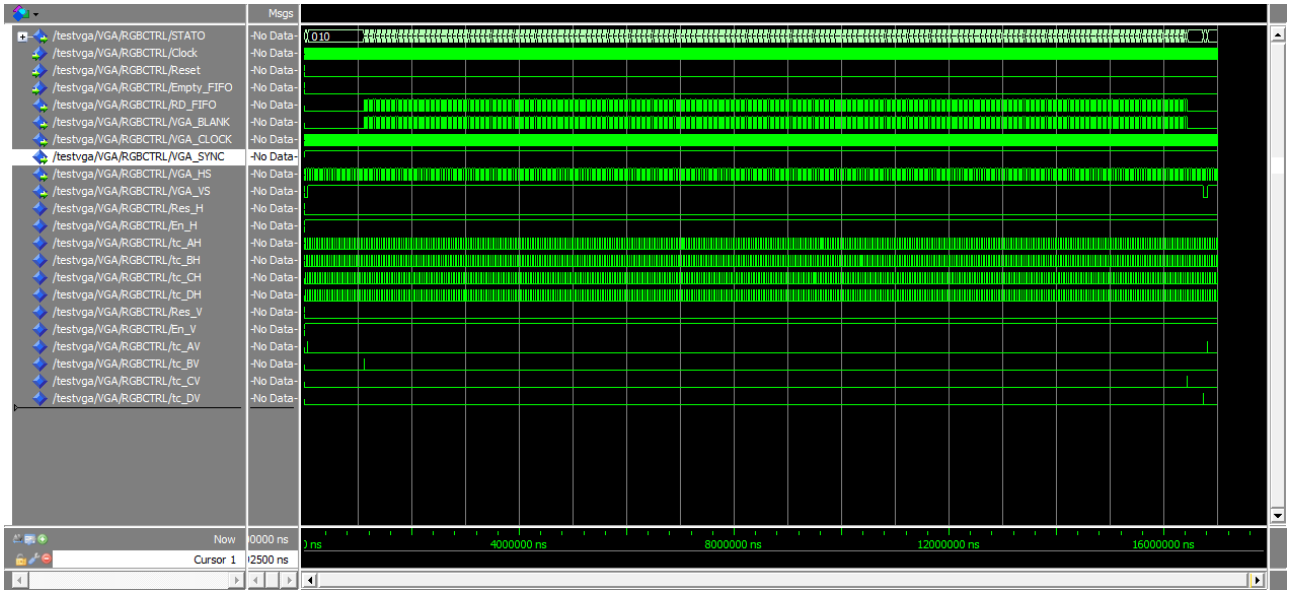
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity TestRGB is
end entity;
architecture behaviour of TestRGB is
component RGBController is
PORT(
Clock,Reset,Empty_FIFO: IN std_logic;
STATO: OUT std_logic_vector( 2 downto 0);
VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS,RD_FIFO: OUT std_logic);
end component;
signal Clock,Reset,Empty_FIFO: std_logic;
signal STATO: std_logic_vector( 2 downto 0);
signal VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS,RD_FIFO: std_logic;
BEGIN
Process
BEGIN
clock<='1';
wait for 20 ns;
clock<='0';
wait for 20 ns;
end process;
Process
BEGIN
```

```
reset<='1';
Empty_FIFO<='1';
wait for 10 ns;
reset<='0';
Empty_FIFO<='0';
wait;
end process;
RGB: RGBController PORT MAP(Clock,Reset,Empty_FIFO,STATO,VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS,RD_FIFO);
end architecture;
```

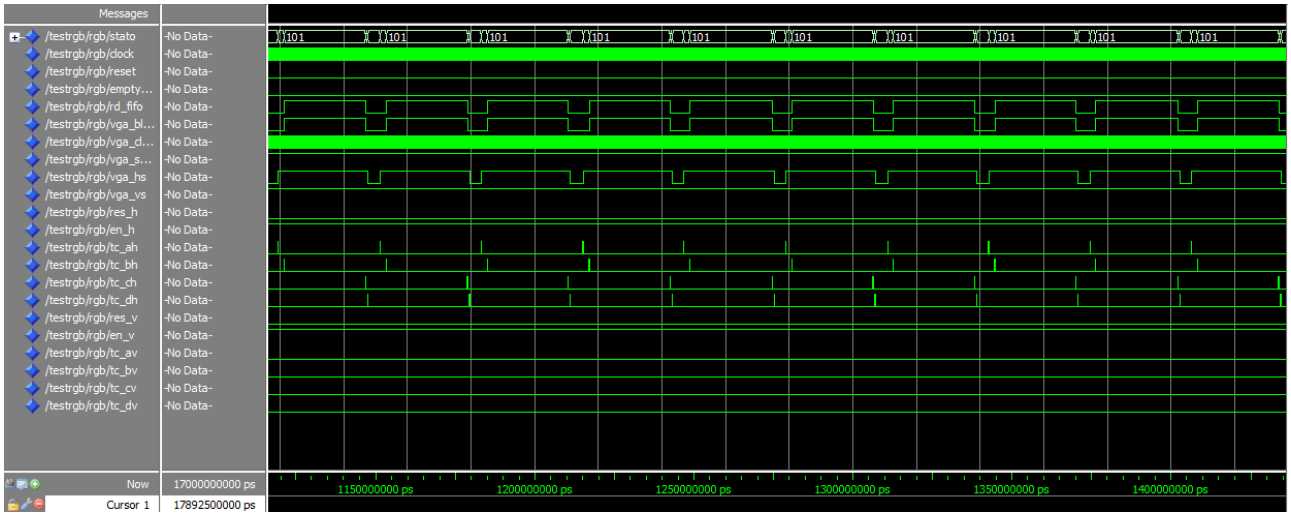
Timing

Nell figure sottostanti vengono mostrati il Timing complessivo ed il Timing dettagliato. Nel primo è raffigurato un ciclo completo del segnale di sincronismo verticale mentre nel secondo alcuni cicli del segnale di sincronismo orizzontale.

Timing Complessivo

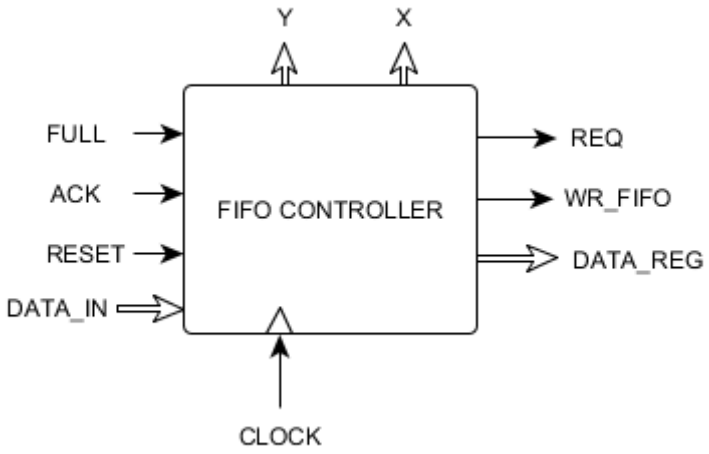


Timing Dettagliato



2.2 FIFO Controller

Il blocco FIFO Controller è presentato di seguito:

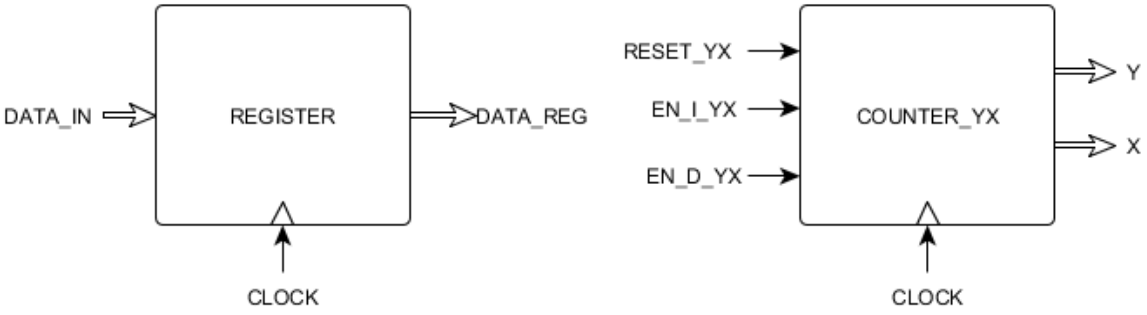


2.2.1 Specifiche

Il blocco FIFO Controller permette di gestire il caricamento dei dati provenienti dalla memoria esterna. Esso deve garantire, che il blocco RGB Controller, abbia sempre dei dati validi da rappresentare sul monitor. È inoltre necessario che, il trasferimento di un dato dalla memoria alla FIFO, impieghi a regime, un colpo di clock. Le informazioni tra memoria e FIFO Controller sono scambiate con un protocollo regolato da Handshake: il segnale di REQ viene asserito durante la richiesta di un dato alla memoria. Vengono inoltre fornite le coordinate del pixel Y.X richiesto. Non appena il Memory Controller ha completato la lettura dalla SRAM esterna, asserisce il segnale ACK, fornendo parallelamente il dato sulla porta DATA_IN. Le richieste dei dati al Memory Controller (o qualsiasi altro blocco esterno) vengono effettuate ogni qualvolta la memoria FIFO non risulta più piena.

2.2.2 Data Path:

Il Data Path del FIFO Controller è costituito da un registro e da un semplice contatore Y-X in grado di fornire, in corrispondenza di ogni richiesta al Memory Controller, le coordinate del pixel da memorizzare all'interno della memoria FIFO.



Counter_YX

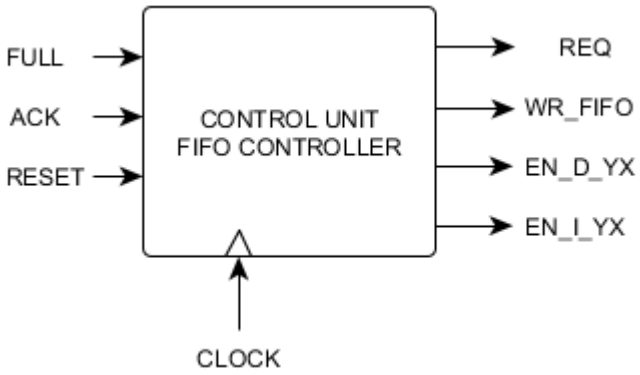
Il contatore **Counter_YX** fornisce le coordinate Y-X di un pixel. Con Y si indica la riga dello schermo mentre con X la colonna. Il campo di variazione è rispettivamente 0-479 e 0-639. L'incremento di Y avviene ogni qualvolta X raggiunge nuovamente il numero 0. Di seguito sono elencati i segnali di I/O:

- RESET_YX: segnale di reset che permette di azzerare il contatore;
- EN_I_YX: segnale che permette di abilitare l'incremento del contatore;
- EN_D_YX: segnale che permette di abilitare il decremento del contatore;
- Y: coordinata Y dello schermo;
- X: coordinata X dello schermo;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

Register

Il registro **Register** è usato dal FIFO Controller per memorizzare temporaneamente il dato proveniente dalla memoria e garantire un trasferimento Pipeline tra memoria e FIFO Controller. Il dato presente su DATA_REG viene scritto all'interno della FIFO solo se considerato valido.

2.2.3 Control Unit



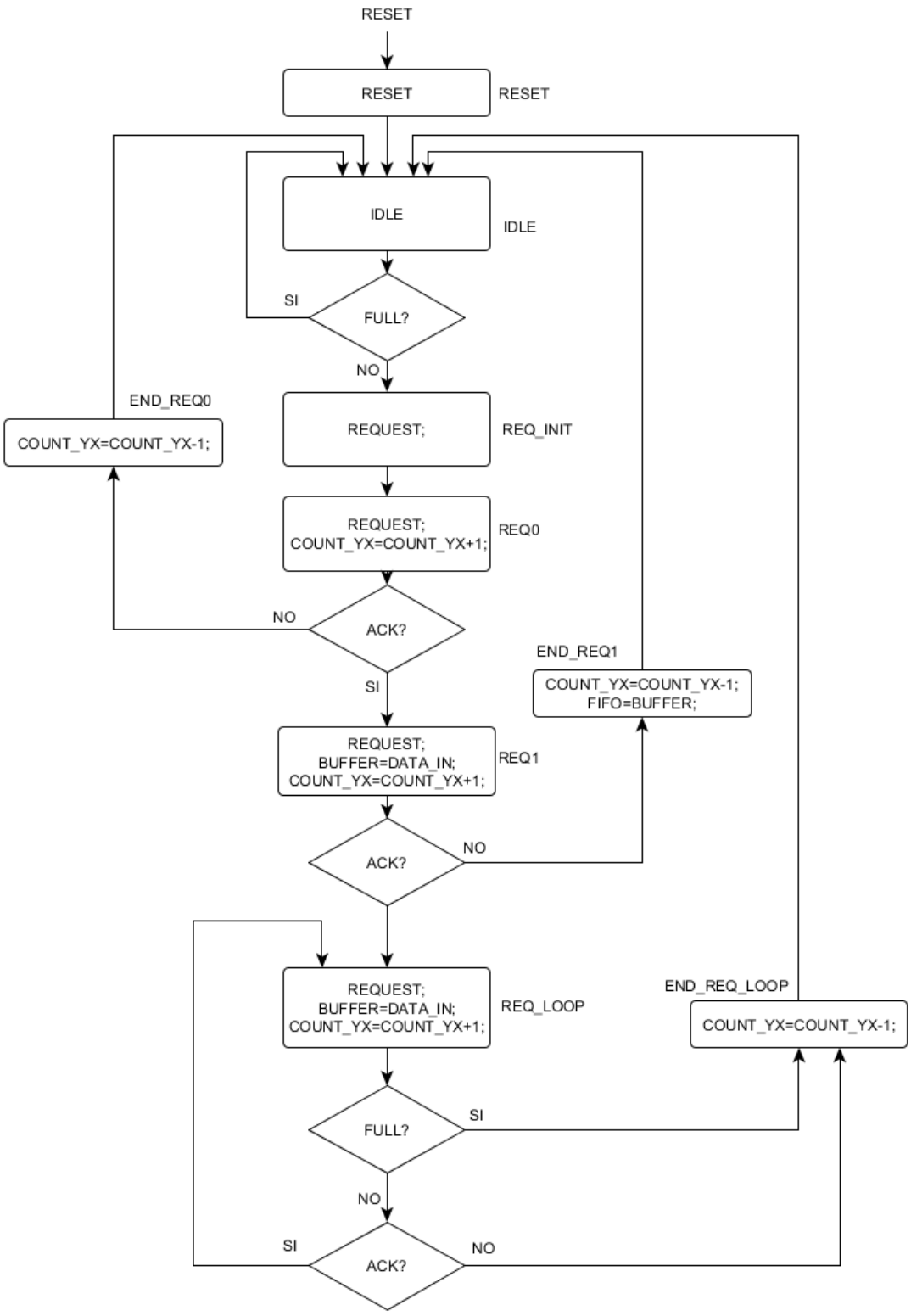
La Control Unit permette di controllare il Data Path per eseguire l'algoritmo richiesto. Essa presenta i seguenti segnali di I/O attivi con livello logico '1':

- FULL: segnale inviato dalla FIFO per indicare che la memoria è piena;
- ACK: segnale di Acknowledge inviato dal Memory Controller dopo aver servito una richiesta del FIFO Controller;
- REQ: segnale di Request per la richiesta di un nuovo dato al Memory Controller;
- EN_D_YX: segnale per abilitare il decremento del contatore **Counter_YX**;
- EN_I_YX: segnale per abilitare l'incremento del contatore **Counter_YX**;
- WR_FIFO: segnale per abilitare la scrittura nella memoria FIFO;
- RESET: segnale per azzerare la macchina a stati;
- CLOCK: ingresso del segnale di temporizzazione;

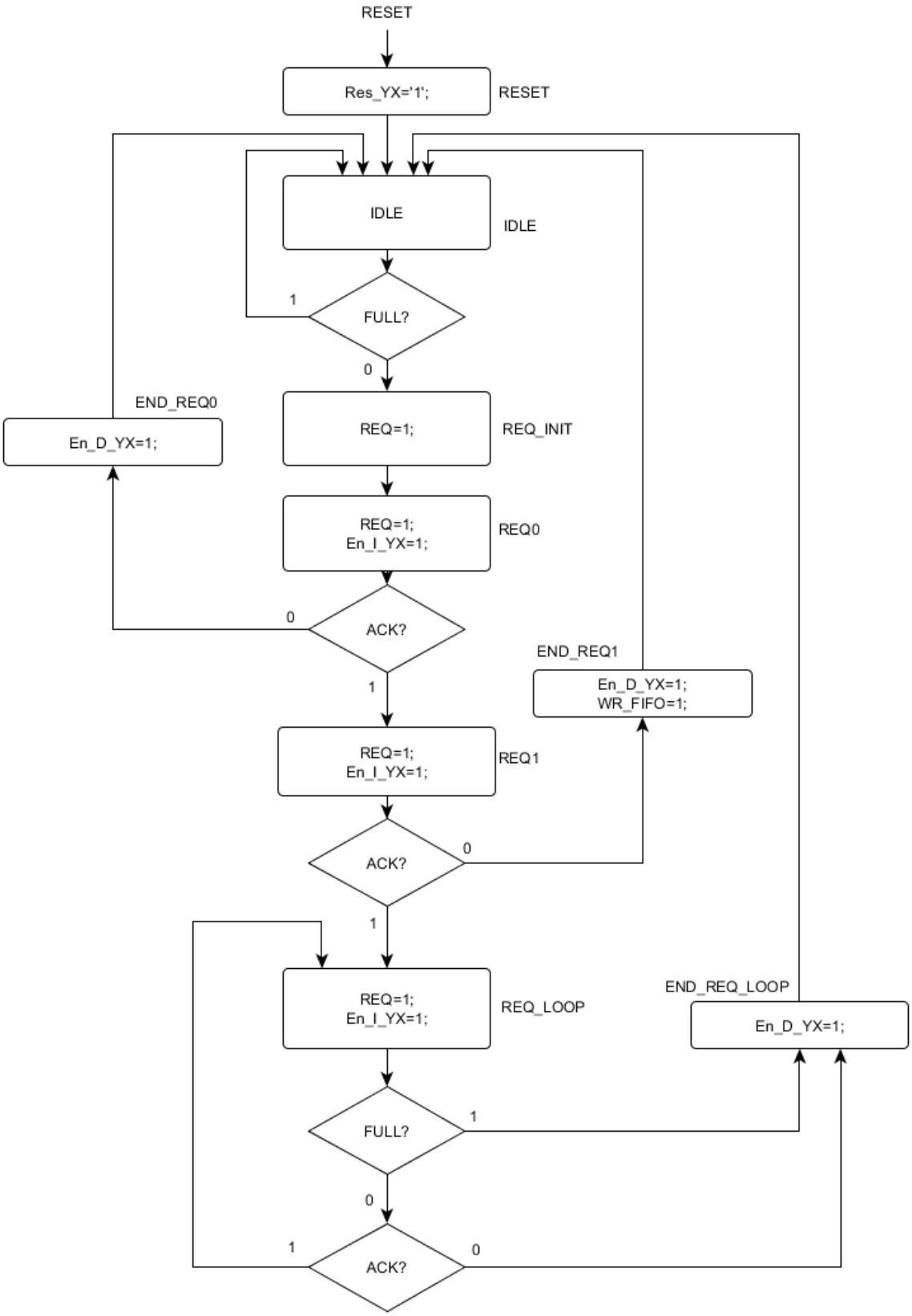
2.2.4 Asm Charts

Di seguito sono mostrati i diagrammi per la realizzazione della macchina a stati. In particolare vengono presentati l'Asm-Data e l'Asm-Control.

Asm Data



Asm Control

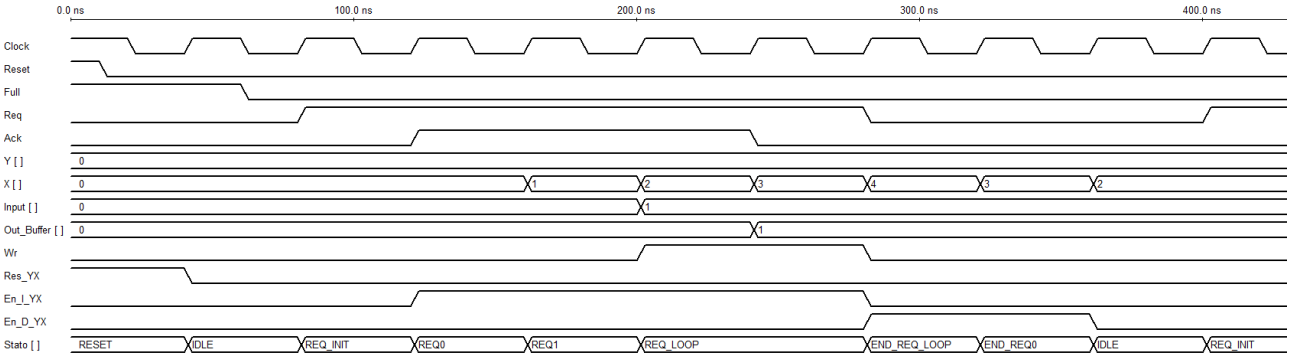

RESET
Res_YX='1';
RESET
IDLE
IDLE
1
FULL?
0
END_REQ0
En_D_YX=1;
REQ=1;
REQ_INIT
REQ=1;
En_I_YX=1;
REQ0
0
ACK?
END_REQ1
En_D_YX=1;
WR_FIFO=1;
1
REQ=1;
En_I_YX=1;
REQ1
ACK?
0
1
REQ=1;
En_I_YX=1;
REQ_LOOP
END_REQ_LOOP
En_D_YX=1;
FULL?
1
0
1
ACK?
0

Nota: nei diagrammi mostrati sono considerati a zero tutti i segnali non indicati all'interno degli stati. Sono stati omessi per rendere più semplice e compatta la rappresentazione dell'algoritmo.

Per permettere la memorizzazione di un dato ogni colpo di clock, il FIFO Controller è stato realizzato con un livello di Pipeline. Il sistema necessita quindi di una latenza iniziale per poter entrare a regime. In particolare sono richiesti 3 colpi di clock.

2.2.5 Timing

Di seguito è mostrato un esempio di transizione degli stati del FIFO Controller.



2.2.6 Simulazione

Grazie all'ausilio del software ModelSim è stato simulato il funzionamento del blocco FIFO Controller. Di seguito è mostrato il codice VHDL del TestBench e il timing ottenuto.

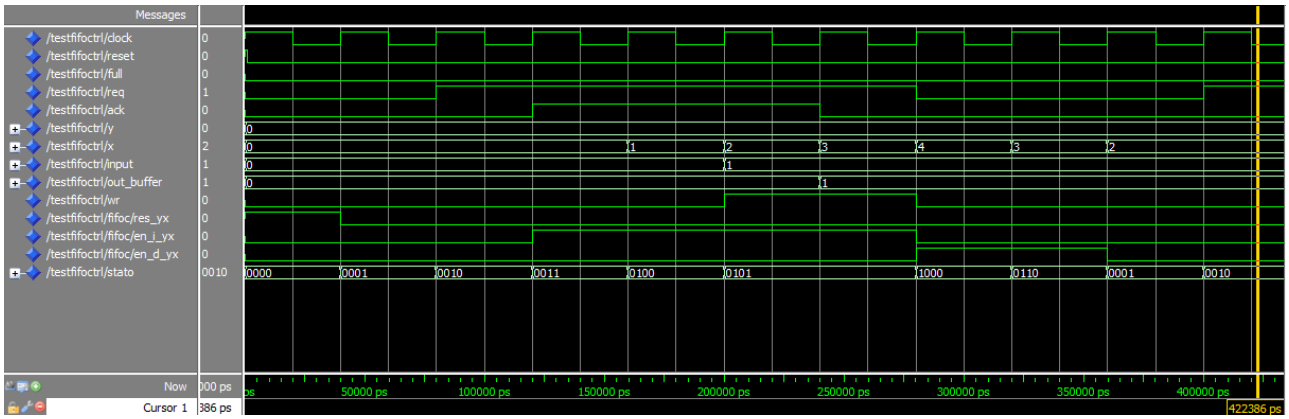
Codice VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity TestFIFOCtrl is
end entity;
architecture behaviour of TestFIFOCtrl is
Component FIFOController is
PORT(
clock,Reset,Full,ACK: IN std_logic;
Y: OUT std_Logic_vector( 8 downto 0 );
X: OUT std_Logic_vector( 9 downto 0 );
STATO: OUT std_logic_vector( 3 downto 0);
DataIn: IN std_logic_vector( 29 downto 0);
Out_Buffer: OUT std_logic_vector( 29 downto 0);
REQ,WR: OUT std_logic);
end component;
type MEMORY is array (0 to 15) of integer;
signal FIFO_MEMORY: MEMORY:=(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
signal clock,Reset,Full,ACK: std_logic;
signal Y: std_Logic_vector( 8 downto 0 );
signal X: std_Logic_vector( 9 downto 0 );
signal STATO: std_logic_vector( 3 downto 0);
signal REQ,WR: std_logic;
signal DataIn: integer:=0;
signal flag: std_logic:='0';
signal Out_Buffer,Input: std_logic_vector( 29 downto 0);
```

```
BEGIN
Process
BEGIN
CLock<='1';
wait for 20 ns;
CLock<='0';
wait for 20 ns;
end process;


Process(Clock,REQ)
BEGIN
if(Clock'EVENT AND CLOCK='1' AND Flag='1')then
ACK<='0';
elsif(Clock'EVENT AND CLOCK='1' AND REQ='1')then
ACK<='1';
DataIn<=FIFO_MEMORY(to_integer(unsigned(X)));
elsif(Clock'EVENT AND CLOCK='1' AND REQ='0')then
ACK<='0';
end if;
end process;
process
begin
Reset<='1';
Full<='0';
wait for 1 ns;
Reset<='0';
wait for 200 ns;
Flag<='1';
wait;
end process;
input<=std_logic_vector(to_unsigned(Datain,30));
FIFOC:  FIFOController PORT MAP(clock,Reset,Full,ACK,Y,X,STATO,Input,Out_Buffer,REQ,WR);
end architecture;
```

Timing

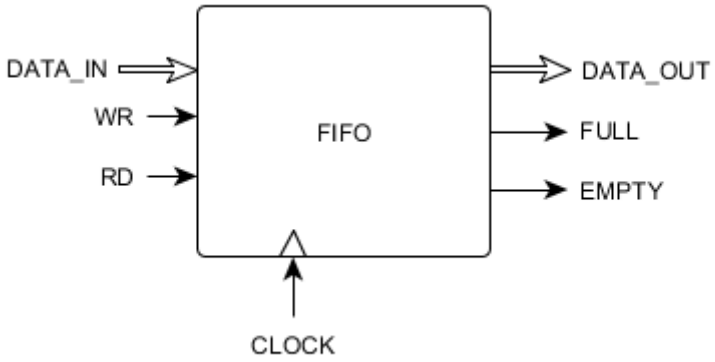
Timing del FIFO Controller.



Come si può notare il funzionamento risulta aderente alle specifiche di progetto.

2.2 Memoria FIFO

La memoria FIFO permette di accodare un certo numero di Pixel inviati successivamente al monitor. In condizioni ideali, la FIFO, non sarebbe necessaria. Infatti, data una richiesta al Memory Controller, esso fornisce in un colpo di clock il rispettivo dato permettendo al sistema di funzionare. Nelle condizioni reali, però il Memory Controller può risultare occupato in operazioni di scrittura di un dato proveniente dalla linea seriale. In questo modo il sistema potrebbe fermarsi. Considerando che, il monitor ha bisogno di un aggiornamento continuo, è necessario inserire una FIFO, creando una coda di dati. Essa viene caricata con nuovi dati ogni qualvolta non risulta piena e il Memory Controller risulta disponibile. Nei momenti in cui quest'ultimo è occupato deve essere garantito che la FIFO non si svuoti completamente causando il blocco del sistema. Considerando che: il trasferimento di un dato tra UART Controller e Memory Controller impiega due colpi di Clock (trasferimento regolato da HandShake), che la memorizzazione nella memoria SRAM asincrona impiega un colpo di clock e che la linea seriale può inviare un dato ogni ms circa è sufficiente usare una FIFO da 16 locazioni. Il numero è in eccesso rispetto alla specifica minima ma permette di guadagnare un margine che garantisce in ogni caso che i dati al convertitore DAC siano validi. Di seguito è mostrata la struttura della memoria ed i segnali di I/O e di controllo.



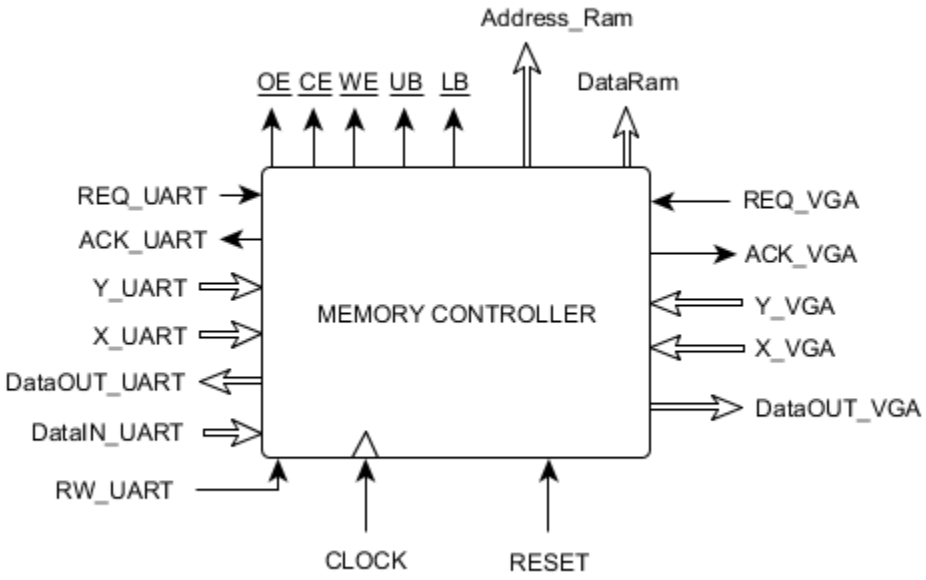
- DATA_IN: ingresso del dato da memorizzare all'interno della FIFO. Parallelismo 30 bit;
- WR: segnale di controllo per abilitare la scrittura sincrona. Attivo con livello logico '1';
- RD: segnale di controllo per abilita la lettura sincrona del dato. Attivo con livello logico '1';
- FULL: segnale di uscita che viene asserito quando la FIFO è piena. In particolare viene portato a livello logico '1' se è rimasta una sola locazione libera;
- EMPTY: segnale di uscita che viene asserito quando la FIFO è vuota;
- DATA_OUT: porta di uscita del dato ricevuto dalla FIFO in seguito ad una lettura;
- CLOCK: segnale di temporizzazione;

2.2.1 Codice VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity FIFO is
GENERIC ( N: integer:=8);
PORT(Reset,Clock,RD,WR: IN std_logic;
DataIN: IN std_logic_Vector( 29 downto 0);
FULL,EMPTY: OUT std_logic;
DataOut: OUT std_logic_Vector( 29 downto 0));
end entity;
architecture behaviour of FIFO is
type MEMORY is array (0 to (N-1)) of std_logic_vector( 29 downto 0);
signal FIFO_MEMORY: MEMORY;
signal CountWR,CountRD,Status: integer:=0;
BEGIN
WRITE:Process(Reset,Clock,WR)
BEGIN
if(Reset='1')then
CountWR<=0;
FIFO_MEMORY<=(others=>(others=>'0'));
elsif(clock'EVENT and clock='1' AND WR='1')then
FIFO_MEMORY(CountWR)<=DataIn;
CountWR<=CountWR+1;
if(CountWR = (N-1))then
CountWR<=0;
end if;end if;
end process;
READ:Process(Reset,Clock,RD)
BEGIN
if(Reset='1')then
CountRD<=0;
elsif(clock'EVENT and clock='1' AND RD='1')then
CountRD<=CountRD+1;
if(CountRD = (N-1))then
CountRD<=0;
end if;end if;
end process;
DataOut<=FIFO_MEMORY(CountRD);
FLAGS: Process(Clock,Reset,RD,WR)
BEGIN
if(Reset='1')then
Status<=0;
Empty<='1';
Full<='0';
elsif(Clock'EVENT AND Clock='1' AND RD='0' AND WR='1' )then
Status<=Status+1;
Empty<='0';
if(Status=N)then
Status<=N;
Full<='1';
elsif(Status=(N-2) OR Status=(N-1))then
Full<='1';
end if;
elsif(Clock'EVENT AND Clock='1' AND RD='1' AND WR='0' )then
Status<=Status-1;
Full<='0';
if(Status=0)then
Status<=0;
elsif(Status=1)then
Empty<='1';
end if;
end if;end process;end architecture;
```

3.0 Memory Controller

Il blocco Memory Controller permette di gestire le richieste proveniente dall'UART Controller o dal VGA Controller per l'accesso alla memoria SRAM asincrona presente sulla scheda DE2 Board. Anche se nel sistema complessivo viene implementata solo la scrittura, in fase di progetto è stata prevista anche la possibilità di effettuare letture da parte dell'UART Controller. Di seguito è rappresentato l'insieme delle porte I/O e dei segnali di controllo:



- REQ_UART: segnale di Request inviato dall'UART Controller quando viene richiesta la scrittura o la lettura di un dato;

- ACK_UART: segnale di Acknowledge inviato dal Memory Controller all'UART Controller. In caso di lettura viene inviato dopo aver campionato gli indirizzi Y_UART, X_UART e mostrato il dato su DataOUT_UART. In caso di scrittura invece, viene asserito dopo aver campionato il dato DataIN_UART e gli indirizzi Y_UART e X_UART;

- Y_UART: ingresso con parallelismo a 9 bit che indica la coordinata Y del pixel richiesto o scritto dall'UART Controller;

- X_UART: ingresso con parallelismo a 10 bit che indica la coordinata X del pixel richiesto o inviato dall'UART Controller;

- DataOUT_UART: porta su cui viene mostrato il dato proveniente dalla SRAM in caso di una richiesta di lettura all'indirizzo rappresentato dalle coordinate Y_UART, X_UART;

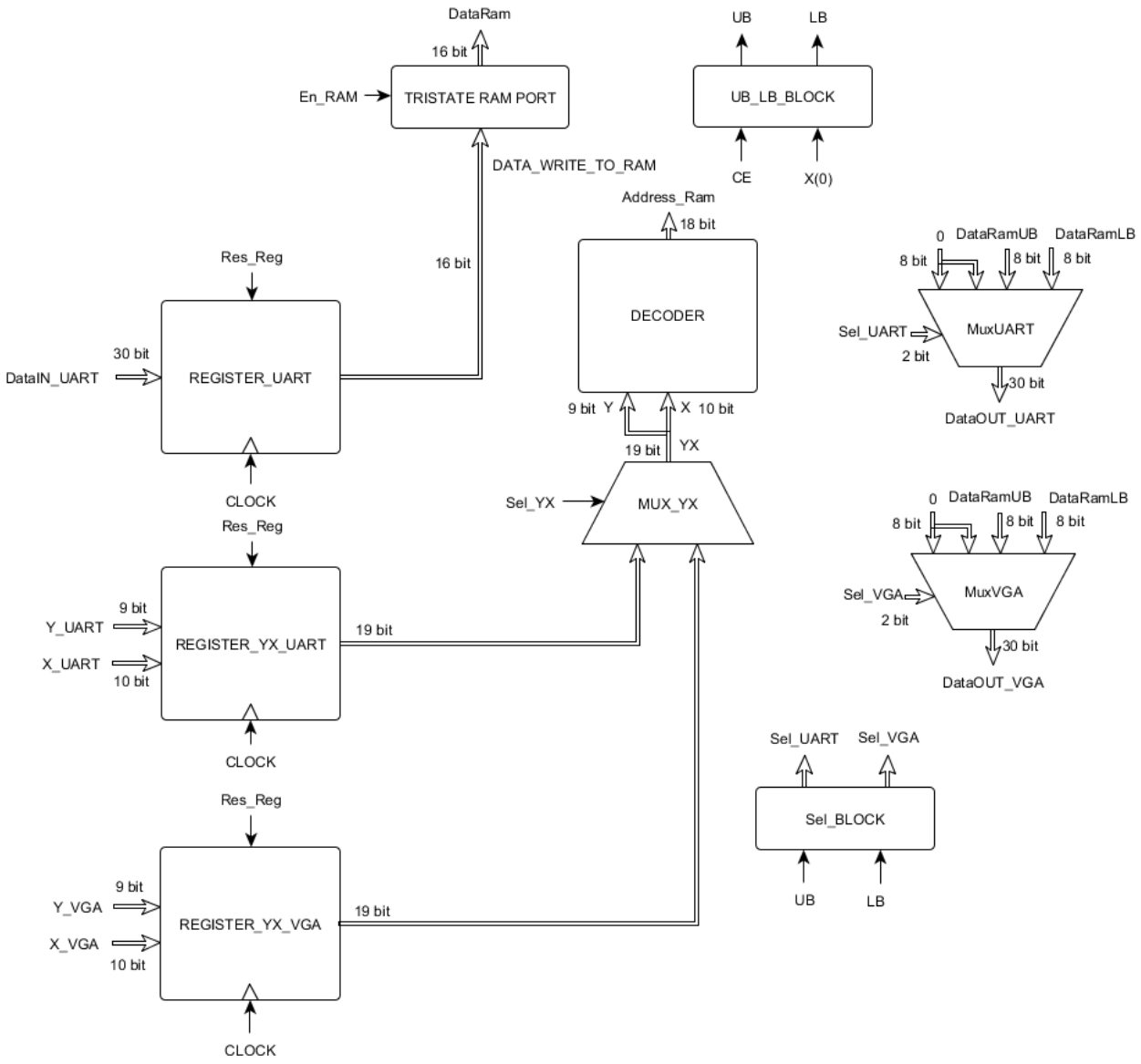
- DataIN_UART: porta su cui viene inviato il dato che l'UART Controller richiede di scrivere all'interno della SRAM all'indirizzo rappresentato dalle coordinate Y_UART, X_UART;
- REQ_VGA: segnale di Request inviato dal VGA Controller quando viene richiesta la lettura di un dato contenuto all'indirizzo rappresentato dalle coordinate Y_VGA, X_VGA;
- ACK_VGA: segnale di Acknowledge inviato dal Memory Controller al VGA Controller dopo aver campionato gli indirizzi Y_VGA, X_VGA e fornito il dato richiesto su DataOUT_VGA;
- Y_VGA: ingresso con parallelismo a 9 bit che indica la coordinata Y del pixel richiesto dal VGA Controller;
- X_VGA: ingresso con parallelismo a 10 bit che indica la coordinata X del pixel richiesto dal VGA Controller;
- DataOUT_VGA: porta su cui viene mostrato il dato proveniente dalla SRAM in caso di una richiesta di lettura all'indirizzo rappresentato dalle coordinate Y_VGA, X_VGA;
- OE: segnale di Output Enable inviato alla SRAM, attivo con livello logico '0';
- CE: segnale di Chip Enable inviato alla SRAM, attivo con livello logico '0';
- WE: segnale di Write Enable inviato alla SRAM, attivo con livello logico '0';
- UB: segnale di Upper Byte inviato alla SRAM, attivo con livello logico '0';
- LB: segnale di Lower Byte inviato alla SRAM, attivo con livello logico '0';
- Address_Ram: indirizzo inviato alla SRAM;
- DataRam: porta bidirezionale connessa alla SRAM parallelismo a 16 bit;
- Clock: segnale di temporizzazione;
- RESET: segnale di Reset che permette di azzerare il Memory Controller;

3.1 Specifiche

Per la realizzazione del Memory Controller è necessario conoscere il Timing per eseguire la lettura e la scrittura sulla SRAM utilizzata. Il modello presente sulla DE2 Board è **IS61LV25616AL.** È inoltre richiesto che le Request inviate dal VGA Controller abbiano precedenza rispetto a quelle del UART Controller per garantire una rappresentazione corretta e continua dei pixel sul Monitor.

3.2 Data Path:

Il Data Path è costituito da una interfaccia con porta tristate connessa ai dati inviati o ricevuti dalla SRAM. Sono inoltre presenti i registri per campionare gli indirizzi e i dati provenienti dall'UART e dal RGB Controller. Infine, è stato inserito un Decoder in grado di convertire le coordinate (Y,X) inviate dall'esterno, in un indirizzo valido per la SRAM. Essa presenta un bus dati a 16 bit. È però possibile scegliere quale byte leggere all'interno di una locazione utilizzando i segnali UB e LB. Quest'ultima contiene due pixel rappresentati su 8 bit. Quando non viene richiesta alcuna lettura ogni porta di uscita del Memory Controller viene forzata al valore 0. La porta DataRam invece viene portata ad alta impedenza se non vengono eseguite operazioni sulla SRAM. Di seguito è mostrato il Data Path completo:



REGISTER_UART

Il registro **REGISTER_UART** viene utilizzato per campionare il dato presente sulla porta DataIN_UART garantendo, in fase di scrittura, un dato stabile. Esso è attivo sul fronte di salita del Clock e presenta un ingresso Res_Reg che permette di azzerarne il contenuto durante lo stato di RESET.

REGISTER_YX_UART

Il registro **REGISTER_YX_UART** viene utilizzato per campionare le coordinate Y e X inviate dall'UART per garantire un indirizzo stabile in ingresso al Decoder durante le fasi di lettura e scrittura. Esso è attivo sul fronte di salita del Clock e presenta un ingresso Res_Reg che permette di azzerarne il contenuto durante lo stato di RESET.

REGISTER_YX_VGA

Il registro **REGISTER_YX_VGA** viene utilizzato per campionare le coordinate Y e X inviate dal VGA Controller per garantire un indirizzo stabile in ingresso al Decoder durante la fase di lettura. Esso è attivo sul fronte di salita del Clock e presenta un ingresso Res_Reg che permette di azzerarne il contenuto durante lo stato di RESET.

TRISTATE RAM PORT

La porta **TRISTATE RAM PORT** è di tipo bidirezionale. Essa permette di leggere i dati inviati dalla SRAM oppure di scrivere al suo interno. Il suo stato è regolato dal segnale En_RAM. Quando viene asserito la porta è un uscita.

MuxUART

Il multiplexer **MuxUART** viene utilizzato per regolare il flusso dei dati sulla porta DataOUT_UART. Durante una lettura richiesta dall'UART Controller, il multiplexer, mostra in uscita il valore del dato fornito dalla SRAM. In particolare, dato che il parallelismo della SRAM è 16 bit, in funzione dell'indirizzo viene mostrata la parte più o meno significativa del dato. Durante le altre operazioni invece, viene l'uscita viene forzata a 0. Il suo stato è regolato dal segnale Sel_UART con parallelismo a 2 bit. Il suo valore dipende dai segnali En_UART, UB ed LB.

MuxVGA

Il multiplexer **MuxVGA** viene utilizzato per regolare il flusso dei dati sulla porta DataOUT_VGA. Durante una lettura richiesta dal VGA Controller, il multiplexer, mostra in uscita il valore del dato fornito dalla SRAM. In particolare, dato che il parallelismo della SRAM è 16 bit, in funzione dell'indirizzo viene mostrata la parte più o meno significativa del dato. Durante le altre operazioni invece, viene l'uscita viene forzata a 0. Il suo stato è regolato dal segnale Sel_VGA con parallelismo a 2 bit. Il suo valore dipende dai segnali En_VGA, UB ed LB.

MuxYX

Il multiplexer **MuxYX** viene utilizzato per scegliere quale indirizzo mandare alla SRAM. Quando Sel_YX è a livello logico '1' l'indirizzo selezionato è quello proveniente dall'UART, in caso contrario dal VGA Controller.

DECODER

Il blocco **DECODER** permette di convertire le coordinate di un pixel (Y,X) in un indirizzo valido per la memoria SRAM. Considerando una risoluzione di 640x480 si ottiene che: Y varia da 0 a 479 mentre X da 0 a 639. Inoltre, ricordando che in ogni locazione della SRAM sono contenuti due pixel, è necessario calcolare l'indirizzo come segue:

Address_Ram = X/2 + Y·320 quando X è pari;

Address_Ram = X/2 + Y·320 quando X è dispari;

Di seguito è mostrato il codice VHDL del blocco:

Codice VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity DecoderAddress is
PORT(
Y: IN std_logic_vector(8 downto 0);
X: IN std_logic_vector(9 downto 0);
Address: OUT std_logic_vector( 17 downto 0));
end entity;
architecture behaviour of DecoderAddress is
signal tempX: std_logic_vector(8 downto 0);
signal tempY: std_logic_vector(17 downto 0);
BEGIN
tempX<=X(9 downto 1);
tempY<= std_logic_vector((to_unsigned(320,9)*unsigned(Y)));
Address<=std_logic_vector(unsigned(tempY)+unsigned("000000000"&tempX));
end architecture;
```

SEL_BLOCK

Il componente **Sel_BLOCK** permette di generare il valore corretto dei segnale Sel_VGA e Sel_MUX in funzione di UB e LB.

Codice VHDL:

```
Sel_UART(0)<= UB AND ( NOT LB ) AND En_UART;
Sel_UART(1)<= (NOT UB) AND LB AND En_UART;
Sel_VGA(0)<= UB AND ( NOT LB ) AND En_VGA;
Sel_VGA(1)<= (NOT UB) AND LB AND En_VGA;
```

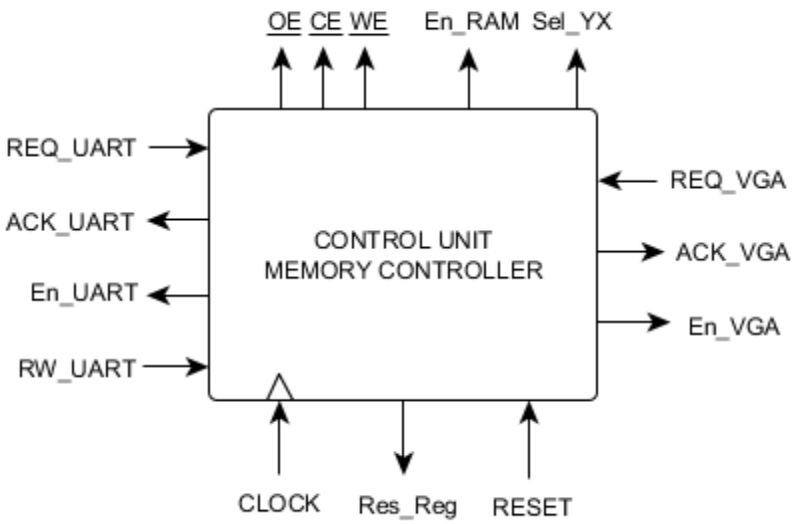
UB_LB_BLOCK

Il blocco **UB_LB_BLOCK** permette di assegnare il valore corretto ai segnali UB ed LB utilizzando il bit meno significativo dell'indirizzo X (pari o dispari) e dello stato del segnale CE. Quando CE è a livello logico '1', UB ed LB sono forzati a '1'.

Codice VHDL:

```
Process(CE,X(0))
BEGIN
if(CE='0')then
LB<= X(0) after (0.001 us);
UB<=  NOT X(0) after (0.001 us);
else
LB<='0';
UB<='0';
end if;
end process;
```

3.3 Control Unit



La Control Unit permette di controllare il Data Path per eseguire l'algoritmo richiesto. Essa presenta i seguenti segnali di I/O:

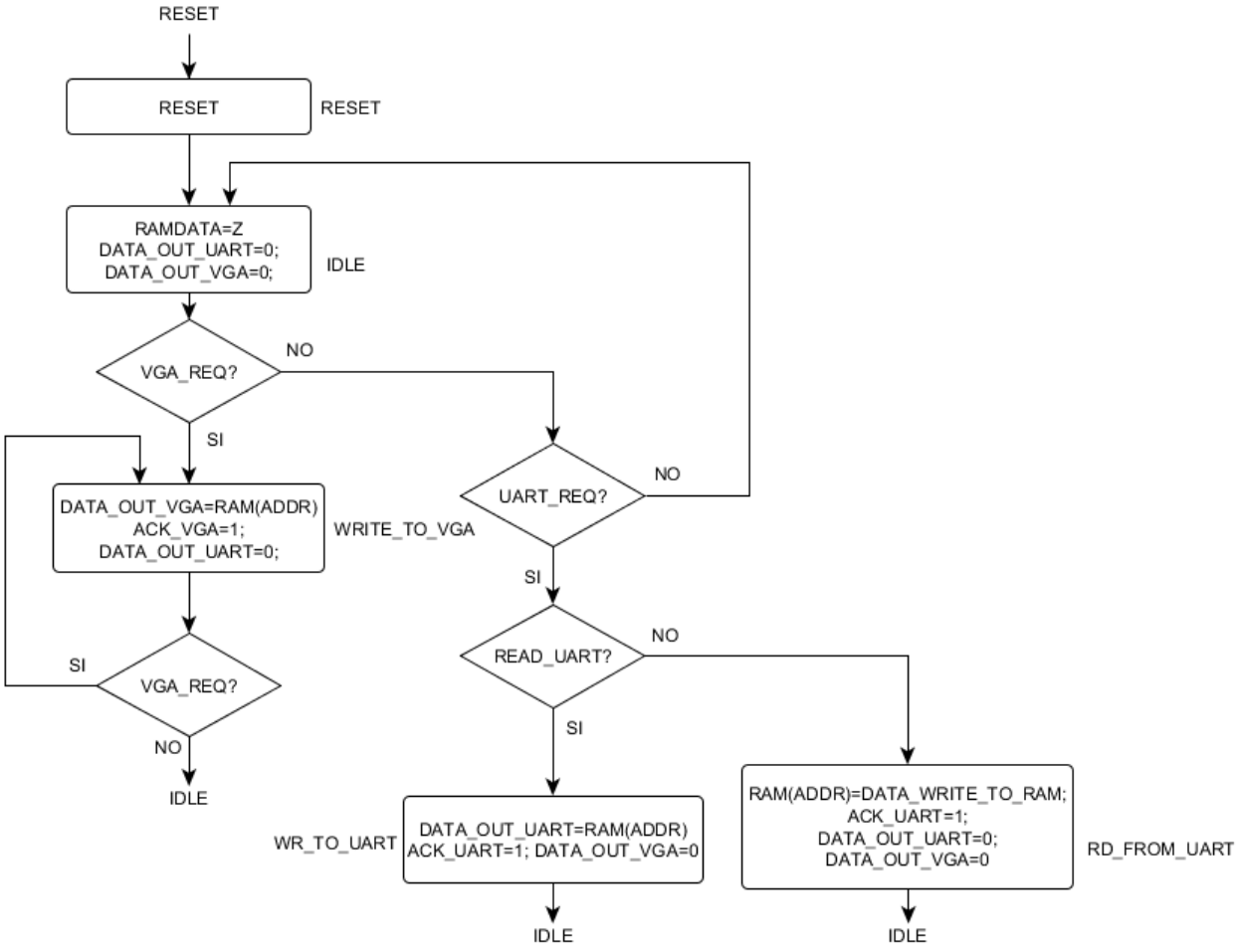
- REQ_UART, ACK_UART: segnali utilizzati per gestire le richieste dell'UART Controller;
- REQ_VGA, ACK_VGA: segnali utilizzati per gestire le richieste del VGA Controller;

- En_UART: segnale asserito quando viene effettuata una lettura dall'UART Controller;
- RW_UART: segnale di ingresso inviato dall'UART Controller per scegliere, in seguito ad una richiesta, se effettuare un operazione di lettura o scrittura. Se asserito viene effettuata una lettura;
- En_VGA: segnale asserito quando viene effettuata una lettura dal VGA Controller;
- OE, CE, WE: segnali di controllo inviati alla SRAM;
- En_RAM: segnale asserito quando la porta DataRam viene utilizzata come uscita;
- Res_Reg: segnale di Reset per i registri interni al Data Path;
- Sel_YX: segnale di controllo che permette di scegliere la coordinata da utilizzare per il calcolo dell'indirizzo della SRAM;
- RESET: segnale per azzerare la macchina a stati;
- CLOCK: ingresso del segnale di temporizzazione;

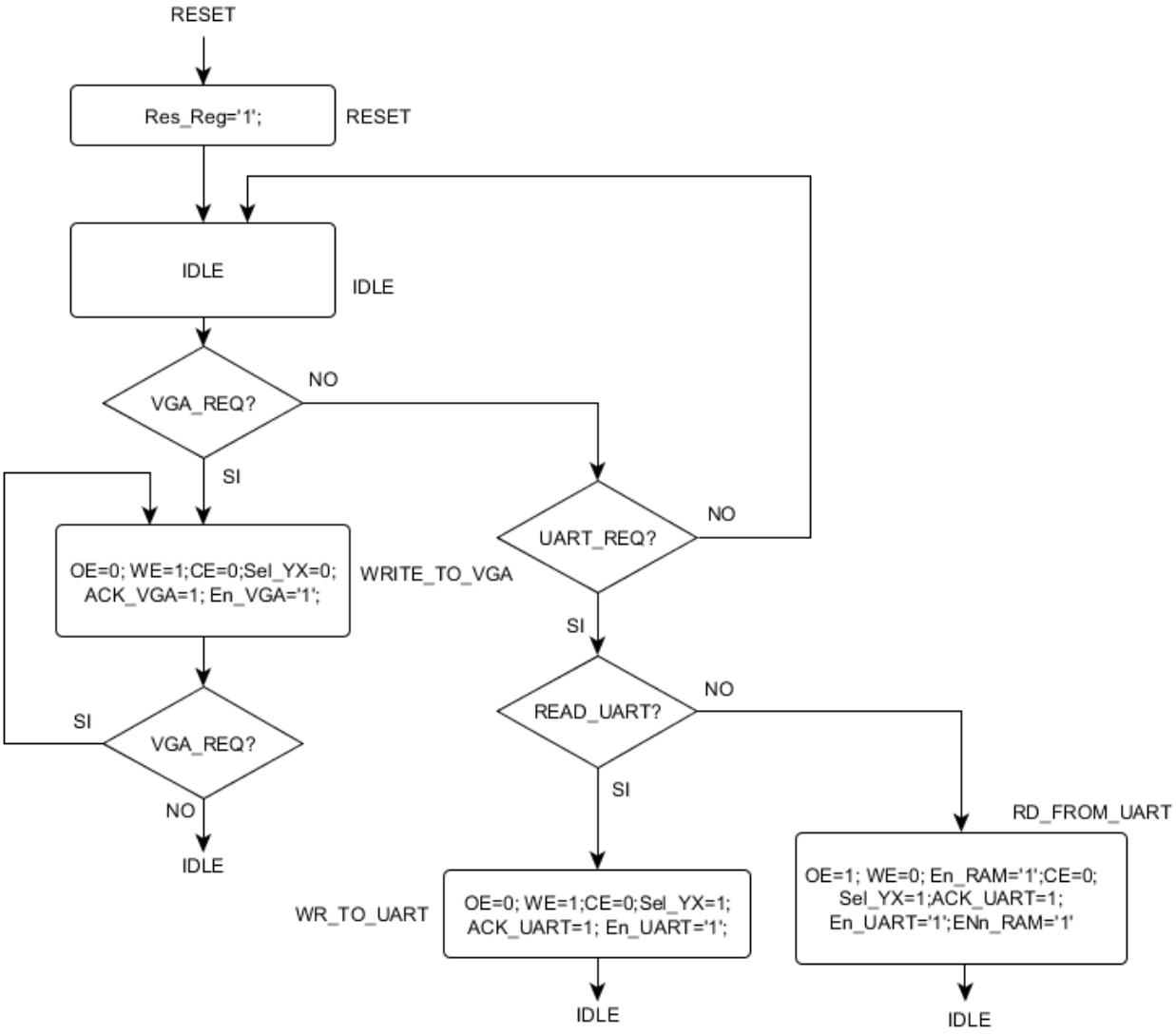
3.4 Asm Charts

Di seguito sono mostrati i diagrammi per la realizzazione della macchina a stati. In particolare vengono presentati l'Asm Data e l'Asm Control.

Asm Data

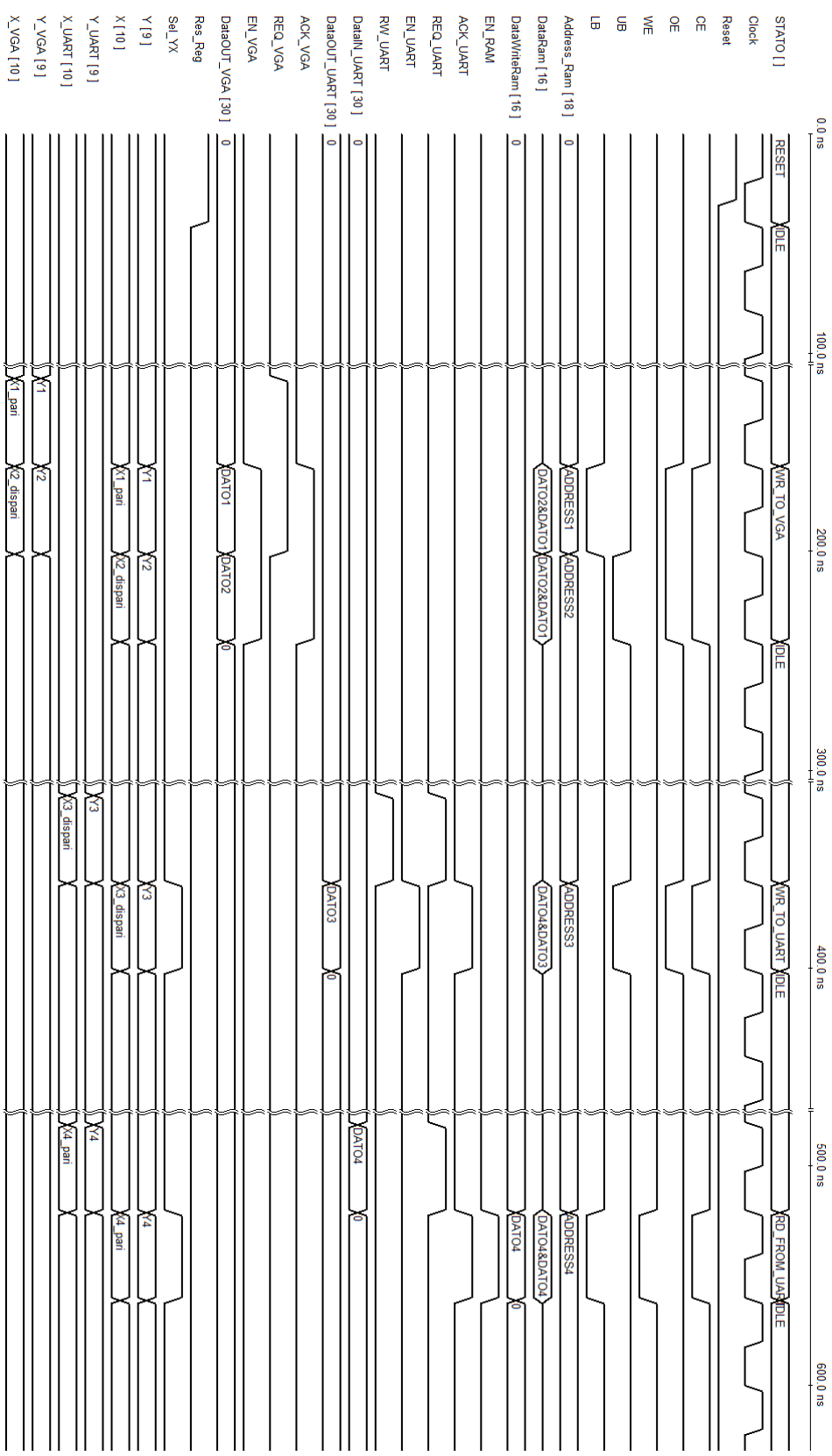


Asm Control


RESET
Res_Reg='1';
RESET
IDLE
IDLE
VGA_REQ?
NO
SI
OE=0; WE=1;CE=0;Sel_YX=0;
ACK_VGA=1; En_VGA='1';
WRITE_TO_VGA
SI
VGA_REQ?
NO
IDLE
UART_REQ?
NO
SI
READ_UART?
NO
SI
WR_TO_UART
OE=0; WE=1;CE=0;Sel_YX=1;
ACK_UART=1; En_UART='1';
IDLE
RD_FROM_UART
OE=1; WE=0; En_RAM='1';CE=0;
Sel_YX=1;ACK_UART=1;
En_UART='1';ENn_RAM='1'
IDLE

3.5 Timing:

Di seguito è mostrato il timing del Memory Controller. Come si può notare sono rappresentati solo gli stati ed i segnali principali.



3.7 Testing:

Attraverso il software ModelSim è stato verificato il corretto funzionamento del blocco Memory Controller. È stata inoltre creata una semplice RAM in grado di rispondere ad un solo indirizzo, quello selezionato dal testbench.

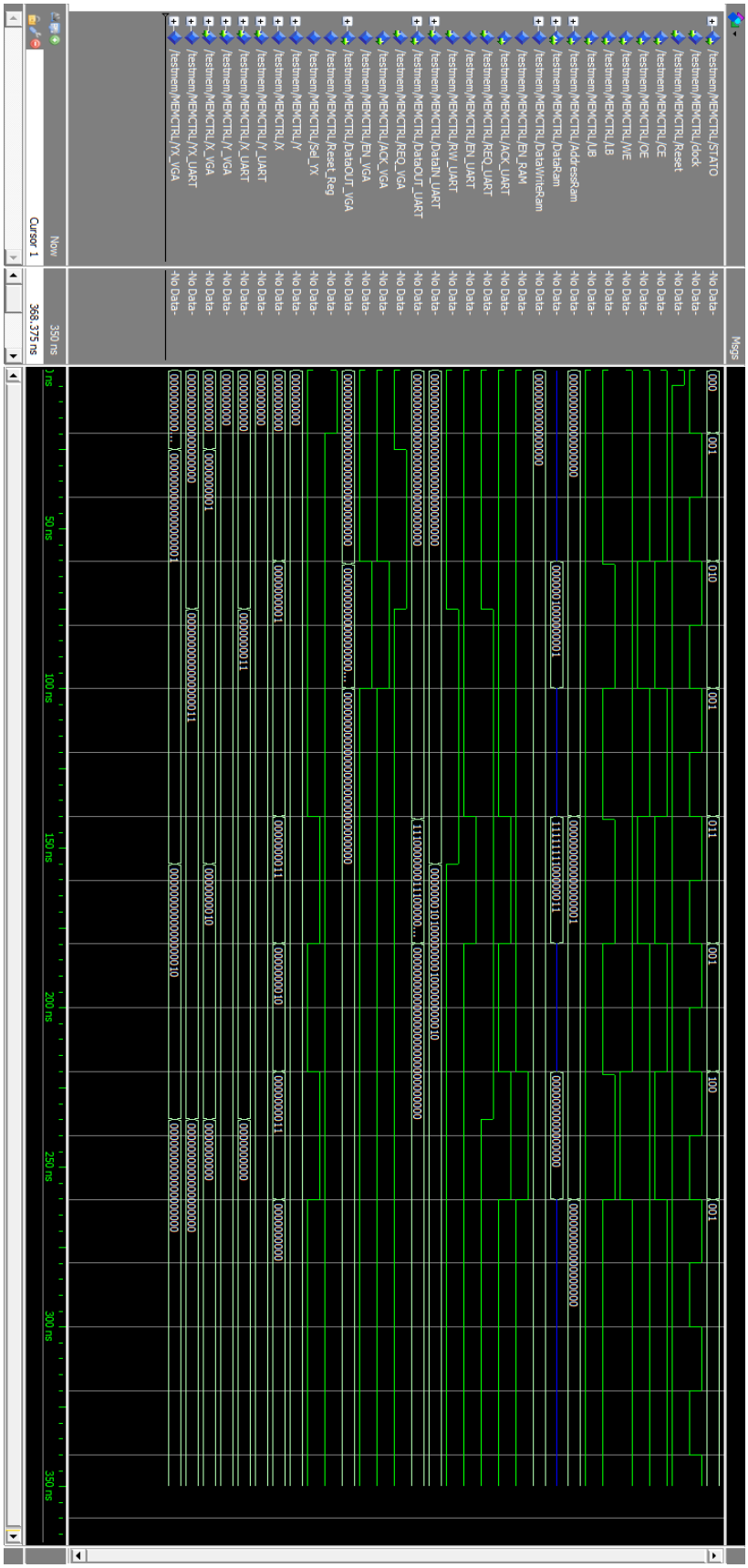
Codice VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity testMEM is
end entity;
architecture behaviour of testMEM is
component MemoryController is
PORT(
clock,Reset: IN std_logic;
CE:BUFFER std_logic;
OE: OUT std_logic;
WE: OUT std_logic;
LB,UB: BUFFER std_logic;
AddressRam: OUT std_logic_vector(17 downto 0);
DataRam: INOUT std_logic_vector( 15 downto 0);
RW_UART,REQ_UART: IN std_logic;
ACK_UART: OUT std_logic;
Y_UART: IN std_Logic_vector( 8 downto 0 );
X_UART: IN std_Logic_vector( 9 downto 0 );
DataIN_UART:  IN std_logic_vector( 29 downto 0);
DataOUT_UART:  OUT std_logic_vector( 29 downto 0);
REQ_VGA: IN std_logic;
ACK_VGA: OUT std_logic;
Y_VGA: IN std_Logic_vector( 8 downto 0 );
X_VGA: IN std_Logic_vector( 9 downto 0 );
DataOUT_VGA: OUT std_logic_vector( 29 downto 0);
STATO: OUT std_logic_vector( 2 downto 0));
end component;
signal clock,Reset:  std_logic;
signal CE: std_logic;
signal OE: std_logic;
signal WE,LB,UB: std_logic;
signal AddressRam: std_logic_vector(17 downto 0);
signal DataRam:  std_logic_vector( 15 downto 0);
signal RW_UART,REQ_UART:  std_logic;
signal ACK_UART:  std_logic;
signal Y_UART:  std_Logic_vector( 8 downto 0 );
signal X_UART: std_Logic_vector( 9 downto 0 );
signal DataIN_UART:  std_logic_vector( 29 downto 0);
signal DataOUT_UART:   std_logic_vector( 29 downto 0);
signal REQ_VGA: std_logic;
signal ACK_VGA: std_logic;
signal Y_VGA: std_Logic_vector( 8 downto 0 );
signal X_VGA: std_Logic_vector( 9 downto 0 );
signal DataOUT_VGA: std_logic_vector( 29 downto 0);
signal STATO: std_logic_vector( 2 downto 0);
BEGIN
Process
BEGIN
clock<='0';
wait for 20 ns;
clock<='1';
wait for 20 ns;
end process;
PRocess(OE,WE,AddressRam)
BEGIN
if(OE='0' AND WE='1' AND AddressRam="00000000000000000")then
DataRam<="0000001000000001";
elsif(OE='0' AND WE='1' AND AddressRam="00000000000000001")then
DataRam<="1111111100000011";
elsif(OE='1')then
DataRam<=(others=>'Z');
```

```
end if;
end process;

Process
BEGIN
reset<='1';
DataIN_UART<="00000000000000000000000000000000";
REQ_UART<='0';
REQ_VGA<='0';
RW_UART<='0';
Y_VGA<=(others=>'0');
X_VGA<=(others=>'0');
Y_UART<=(others=>'0');
X_UART<=(others=>'0');
wait for 5 ns;
Reset<='0';
wait for 20 ns;
REQ_VGA<='1';
Y_VGA<=(others=>'0');
X_VGA<="0000000001";
wait for 50 ns;
REQ_UART<='1';
REQ_VGA<='0';
Y_UART<=(others=>'0');
X_UART<="0000000011";
RW_UART<='1';
wait for 80 ns;
REQ_UART<='1';
REQ_VGA<='0';
Y_VGA<=(others=>'0');
X_VGA<="0000000010";
DataIN_UART<="00000001010000000100000000010";
RW_UART<='0';
wait for 80 ns;
REQ_UART<='0';
Y_VGA<=(others=>'0');
X_VGA<=(others=>'0');
Y_UART<=(others=>'0');
X_UART<=(others=>'0');
wait;
end process;

MEMCTRL: MemoryController PORT MAP(clock,Reset,CE,OE,WE,LB,UB,AddressRam,
DataRam,RW_UART,REQ_UART,ACK_UART,Y_UART,X_UART,DataIN_UART,DataOUT_UART,REQ_VGA,
ACK_VGA,Y_VGA,X_VGA,DataOUT_VGA,STATO);


end architecture;
```

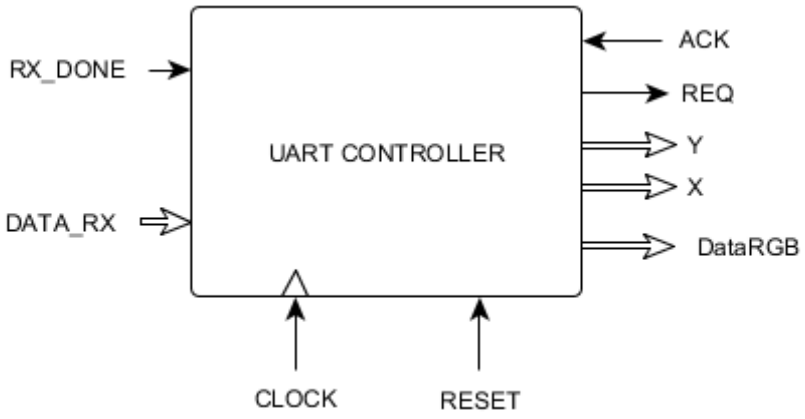
Timing

Di seguito è mostrato il timing ottenuto.



4.0 Progetto UART Controller

Il blocco UART Controller permette di gestire i dati ricevuti dall'UART e inviarli al Memory Controller. Di seguito è rappresentato l'insieme delle porte I/O e dei segnali di controllo:



- RX_DONE: segnale attivato dal ricevitore UART dopo la ricezione di un dato. Durante la ricezione è a livello logico '0';
- DATA_RX: ingresso del dato proveniente dal ricevitore UART. Parallelismo a 8 bit;
- ACK: segnale di Acknowledge proveniente dal Memory Controller in seguito ad una richiesta inviata dall'UART Controller;
- REQ: segnale di Request inviato dal UART Controller per l'invio di un Pixel al Memory Controller;
- Y: uscita con parallelismo a 9 bit che indica la coordinata Y del pixel inviato al Memory Controller;
- X: uscita con parallelismo a 10 bit che indica la coordinata X del pixel inviato al Memory Controller;
- DataRGB: dato inviato al Memory controller contenente i colori del Pixel nel formato R-G-B;

4.1 Specifiche

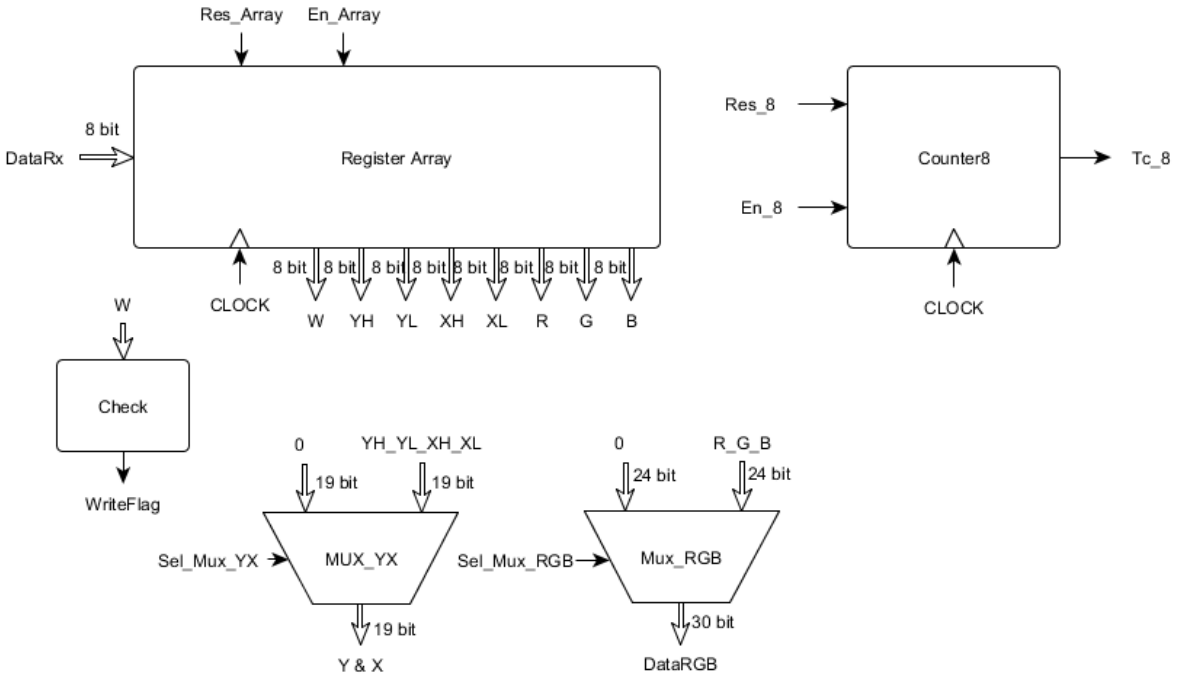
Per progettare l'UART Controller è stato necessario utilizzare un protocollo per permettere all'utente di inviare il Pixel da rappresentare sullo schermo. Di seguito sono mostrati in successioni i byte da inviare attraverso la linea seriale:

1. Carattere "W";
2. Coordinata YH contenente nel LSB, il bit meno significato della coordinata Y;
3. Coordinata YL contenente gli 8 bit meno significativi della coordinata Y;
4. Coordinata XH contenente nei due LSB, i bit meno significativi della coordinata X;
5. Coordinata XL contenente gli 8 bit meno significativi della coordinata X;
6. Colore Rosso su 8 bit;
7. Colore Verde su 8 bit;
8. Colore Blu su 8 bit;

Grazie alla sequenza di questi 8 byte è possibile inviare un pixel memorizzandolo, attraverso il Memory Controller, all'interno della SRAM esterna. I dati scambiati tra il ricevitore UART e l'UART Controller sono regolati dallo stato RX_Done. Il suo stato è a livello logico '1' sia durante lo stato di IDLE che dopo la ricezione di un dato. Si trova a livello logico '0' durante la ricezione. Pertanto, il blocco UART Controller, per capire se il dato fornito dall'UART è valido deve rilevare una transizione 0-1 di Rx_Done. Per il trasferimento dei dati al Memory Controller invece viene utilizzato un protocollo regolato da Handshake. Quando l'UART Controller ha ricevuto gli 8 byte richiesti, effettua una Request al Memory Controller, il quale dopo aver memorizzato il Pixel, asserisce la linea ACK.

4.2 Data Path:

Il Data Path è costituito da un insieme di registri che permettono di memorizzare la sequenza di byte richiesta dal protocollo. Essi sono riuniti in un Array che realizza uno ShiftRegister tra i Registri. Il blocco è indicato con il nome Register Array. È inoltre presente un blocco combinatorio in grado di verificare se nel registro W è presente il carattere 'W', al termine della ricezione degli 8 byte. Il conteggio dei byte ricevuti avviene grazie al contatore Count8. Infine, per evitare che il Memory Controller possa vedere sulle porte DataRGB, Y e X gli stati intermedi durante la memorizzazione, sono stati inseriti dei multiplexer che forzano le uscite a zero.



Register Array

L'insieme dei registri che permettono la memorizzazione degli 8 byte richiesti dal protocollo è riunito in una struttura a ShiftRegister, dove l'uscita del registro B è collegata in ingresso al registro G, l'uscita del registro G è collegata in ingresso al registro R ecc. È presente un segnale di abilitazione alla scrittura che permette di cambiare contemporaneamente il contenuto di tutti i registri nella direzione B to W. È stato infine inserito un ingresso di Reset per azzerare il loro contenuto. Il segnale di Clock è utilizzato per il caricamento sincrono ed è attivo sul fronte positivo.

Check

Il blocco **Check** permette attraverso la linea di uscita WriteFlag di rilevare la presenza del carattere 'W' all'interno del Registro W. Quando è presente, l'uscita è forzata a livello logico '1'.

MuxYX

Il multiplexer **MuxYX** viene utilizzato per mostrare in uscita l'indirizzo Y.X in cui memorizzare il pixel ricevuto dalla linea seriale. Se non vi sono dati da inviare l'uscita viene forzata a zero.

MuxRGB

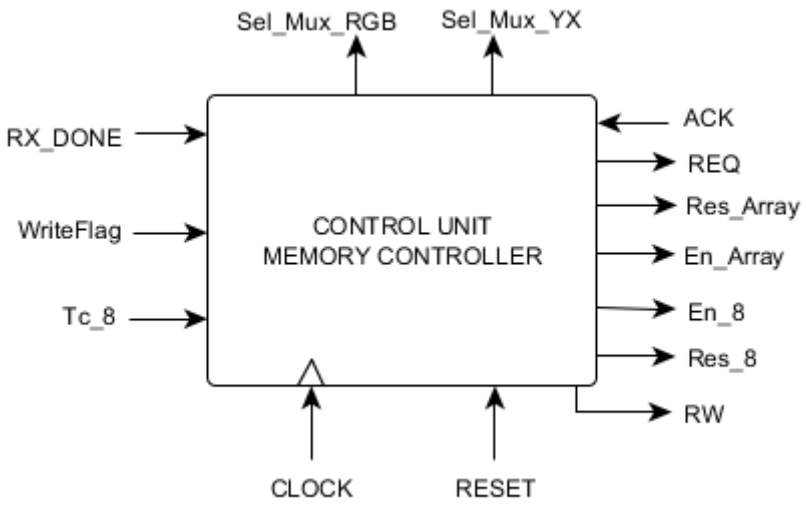
Il multiplexer **MuxRGB** viene utilizzato per mostrare il dato RGB da inviare al Memory Controller. Se non vi sono dati da inviare l'uscita viene forzata a zero.

Counter8

Il contatore **Counter8** permette di contare il numero di byte ricevuti dal ricevitore UART. Quando il Terminal Count TC_8 è attivo, l'UART Controller invia il dato al Memory Controller. Di seguito sono elencati i segnali di I/O:

- RES_8: segnale di reset che permette di azzerare il contatore;
- EN_8: segnale che permette di abilitare l'incremento del contatore;
- TC_8: Terminal Count del contatore. Attivo con livello logico '1' non appena il contatore raggiunge il numero 7;
- CLOCK: ingresso del segnale di temporizzazione attivo sul fronte positivo;

4.3 Control Unit



Di seguito sono mostrati i segnale di controllo della Control Unit:

- Reset: ingresso del segnale di Reset usato per portare la macchina a stati nello stato di RESET;

- RW: segnale di uscita inviato al Memory Controller quando è presente una richiesta di scrittura. È attivo con livello logico ‘0’;
- Res_8: segnale di Reset per azzerare il contatore **Counter8**;

- En_8: segnale di abilitazione per il contatore **Counter8**;

- En_Array: segnale per abilitare il caricamento del dato DataRX all’interno del **Register Array**;

- Res_Array: segnale per azzerare il contenuto del **Register Array**;

- REQ: segnale asserito dall’UART Controller quando viene richiesta una scrittura al Memory Controller;
- ACK: segnale asserito dal Memory Controller quando viene servita una richiesta dell’UART Controller;

- Sel_Mux_YX: segnale di selezione per il multiplexer **MuxYX**;

- Sel_Mux_RGB: segnale di selezione per il multiplexer **MuxVGA**;

- RX_DONE: segnale ricevuto dal ricevitore UART. Esso è a livello logico ‘1’ quando è stato ricevuto un dato e a livello logico ‘0’ durante la ricezione;

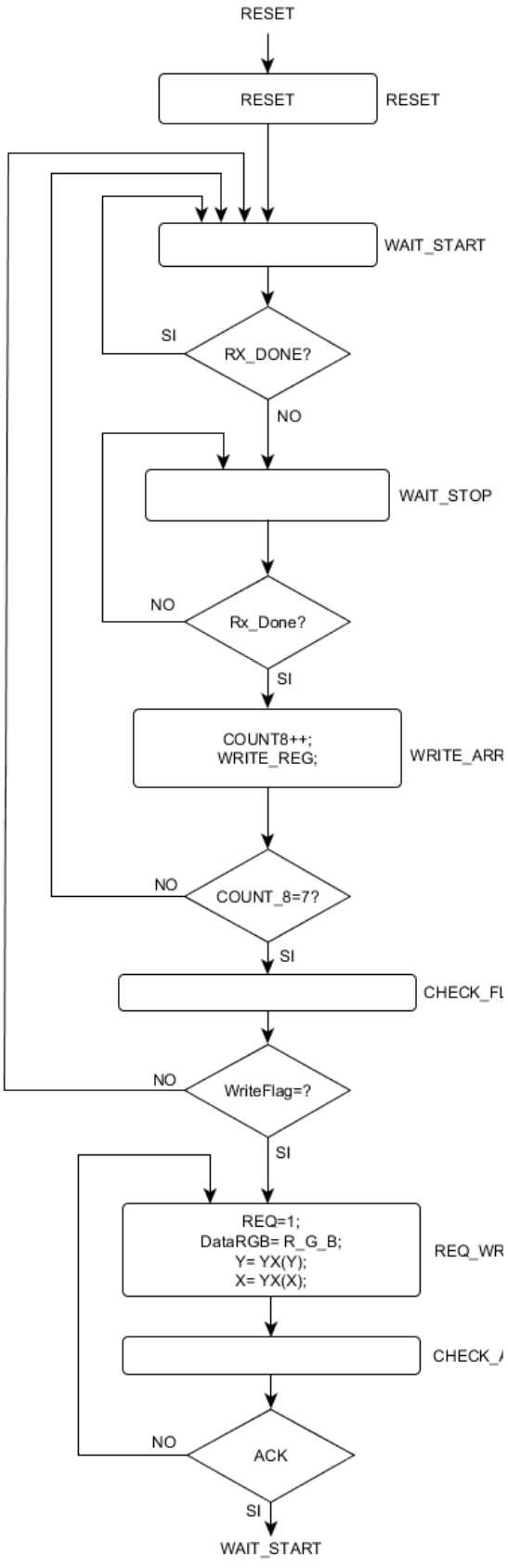
- WriteFLag: segnale ricevuto dal blocco Check che permette di stabilire se all’interno del **Register Array**, nella posizione W, è contenuto il carattere ‘W’;

- TC_8: segnale di Terminal Count del contatore **Counter8**;

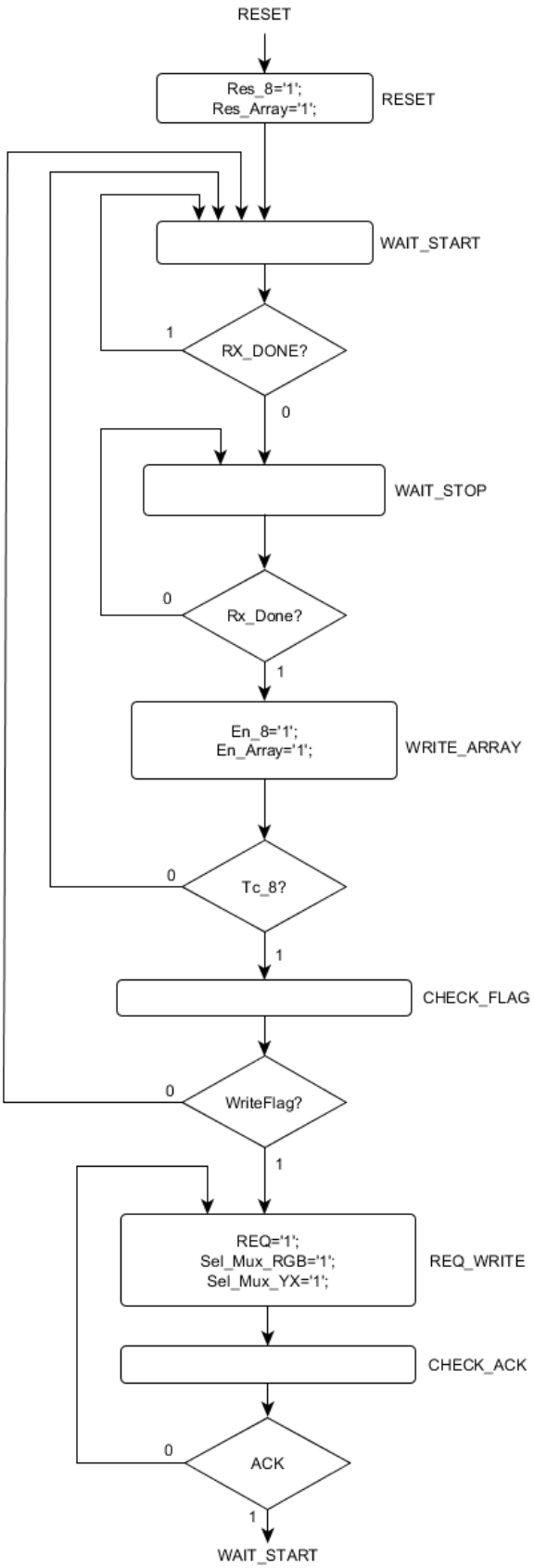
- CLOCK: segnale di temporizzazione;

4.4 Asm Chart

Asm Data

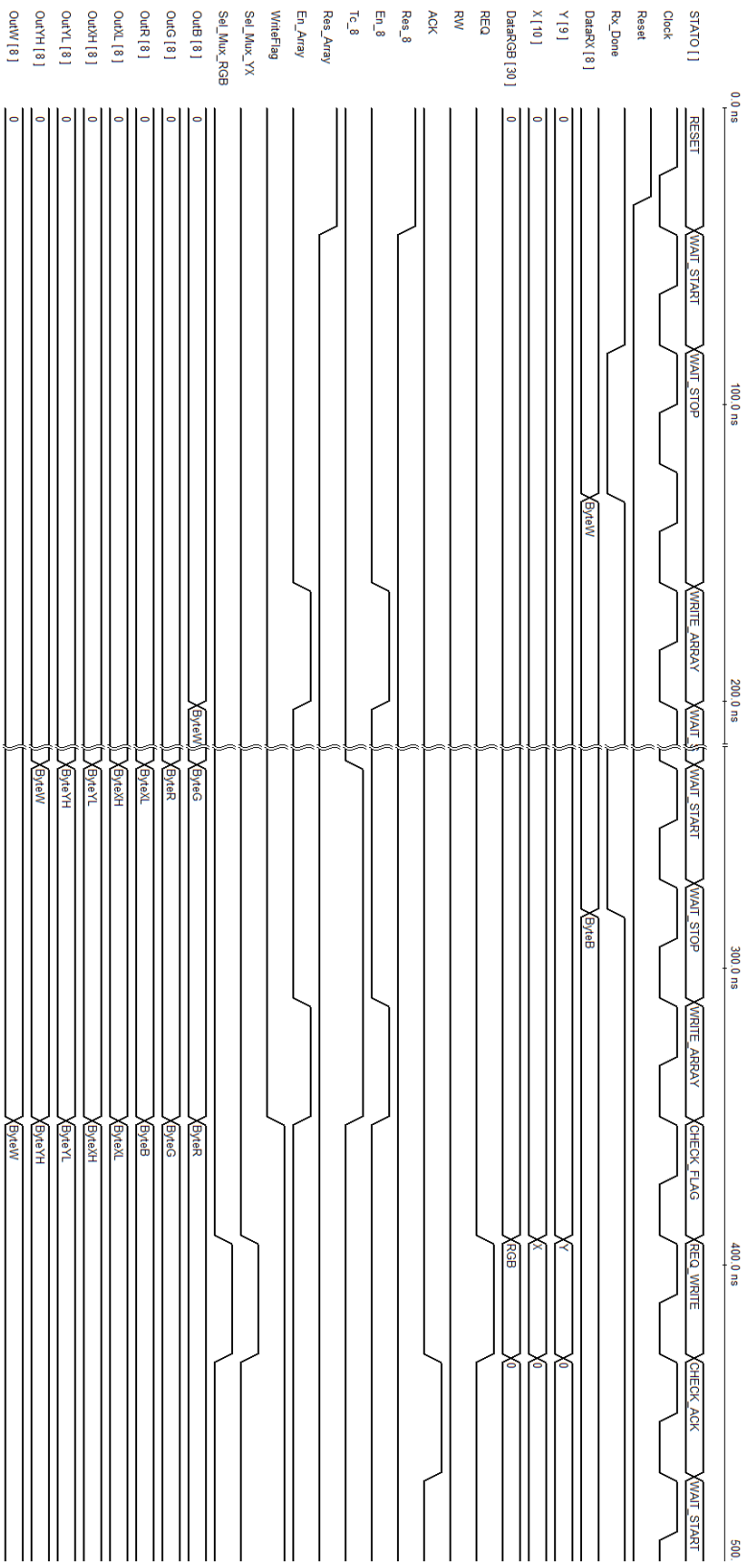


Asm Control



4.5 Timing

Di seguito è mostrato il Timing dell'UART Controller rappresentando in modo compatto la ricezione di uno stream di 8 Byte dal ricevitore UART.



4.6 Simulazione

Attraverso il software ModelSim è stato simulato il funzionamento dell'UART Controller. Di seguito è mostrato il codice VHDL ed il Timing ottenuto.

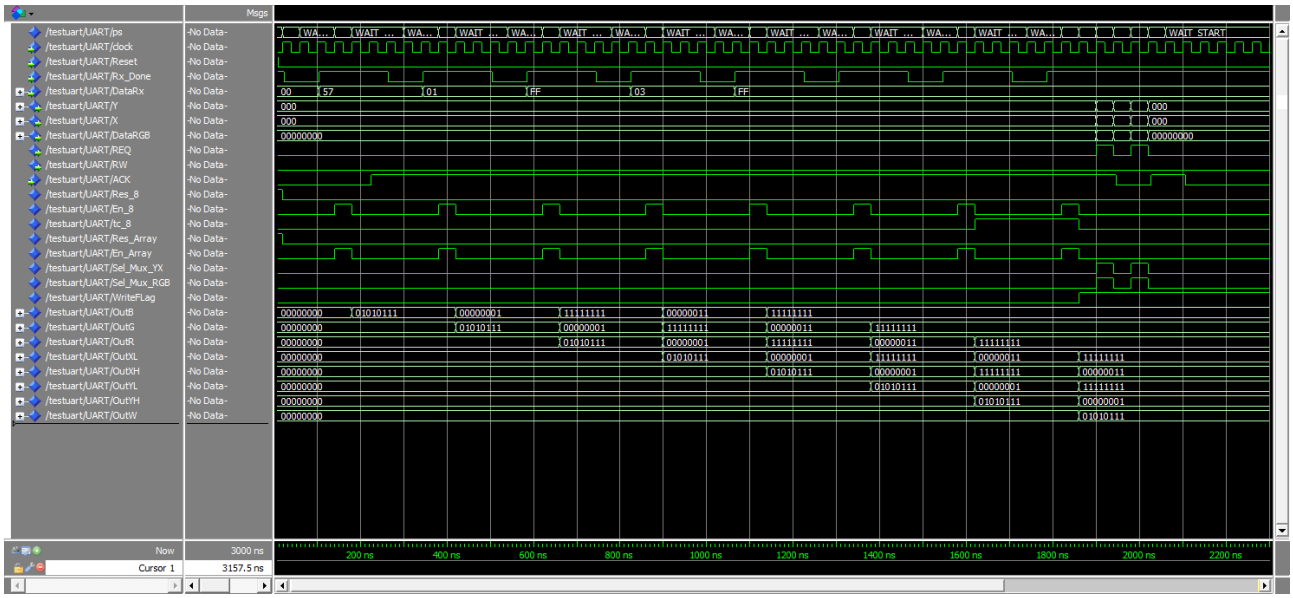
Codice VHDL Simulazione:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity testUART is
end entity;
architecture behaviour of testUART is
component UARTController is
PORT(
clock,Reset,Rx_Done: IN std_logic;
DataRx: IN std_logic_vector( 7 downto 0);
Y: OUT std_Logic_vector( 8 downto 0 );
X: OUT std_Logic_vector( 9 downto 0 );
REQ,RW: OUT std_logic;
ACK: IN std_logic;
DataRGB: OUT std_logic_vector( 29 downto 0));
end component;
signal clock,Reset,Rx_Done: std_logic;
signal DataRx: std_logic_vector( 7 downto 0);
signal Y: std_Logic_vector( 8 downto 0 );
signal X:  std_Logic_vector( 9 downto 0 );
signal REQ,RW:  std_logic;
signal ACK: std_logic;
signal DataRGB: std_logic_vector( 29 downto 0);
BEGIN
Process
BEGIN
Clock<='0';
wait for 20 ns;
CLock<='1';
wait for 20 ns;
end process;
Process
BEGIN
Reset<='1';
Rx_Done<='1';
ACk<='0';
DataRx<=(others=>'0');
wait for 10 ns;
Reset<='0';
wait for 15 ns;
--------- INVIO W
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="01010111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
--------- INVIO YH
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="00000001";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO YL
```

```
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="11111111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO XH
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="00000011";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO XL
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="11111111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO R
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="11111111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO G
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="11111111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
---------INVIO B
Rx_Done<='0';
wait for 80 ns;
Rx_Done<='1';
DataRx<="11111111";
wait for 120 ns;
Ack<='1';
Wait for 40 ns;
Ack<='0';
wait for 80 ns;
Ack<='1';
wait for 80 ns;
Ack<='0';
wait;
end process;
UART: UARTCOntroller PORT MAP(Clock,Reset,RX_DONE,DataRX,Y,X,REQ,RW,ACK,DataRGB);
end architecture;
```

Timing

Di seguito è rappresentato il Timing ottenuto con la simulazione. I risultati sono secondo le aspettative.



5.0 Test del Sistema Complessivo

Per garantire le massime prestazioni nel protocollo VGA è richiesto l'utilizzo di un clock a 25.175 MHz. Per questo motivo è stato implementato, attraverso gli IP MegaBlock interni a Quartus, un PLL in grado di ricavare da 27 MHz (clock presente sulla scheda DE2 Board) la frequenza desiderata. Tutti i blocchi sono stati collegati quindi medesimo clock. L'unica eccezione riguarda il ricevitore UART. Esso è stato progettato considerando un segnale di temporizzazione di 25 MHz. Utilizzandolo a 25.175 MHz è sufficiente modificare il modulo contatore Counter_163 da 163 a 164. Di seguito sono mostrate la simulazione e il test in laboratorio.

5.1 Simulazione

Utilizzando il software ModelSim è stato simulato il sistema complessivo inviando al ricevitore seriale lo stream 'W', Y_H=0, Y_L=0, X_H=0, X_L=2, R=255, G=255 e B=255. L'invio impiega circa 10 ms pertanto dopo un ciclo di rinfresco del monitor pari a circa 17 ms deve apparire in uscita alla coordinata 0.2 il pixel di valore 255.255.255. Per semplificare la simulazione è stata realizzata una SRAM in grado di memorizzare solo all'interno della locazione equivalente alla coordinata 0.2. In tutte le altre celle il contenuto è pari a 0. DI seguito è riportato il codice VHDL del TestBench e il risultato ottenuto.

Codice VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity testSystem is
end entity;
architecture test of testSystem is
component System is
PORT(
Start: IN Std_logic;
Flag2: OUT std_logic;
Rx_Line: IN std_logic;
Clock_27,Clock_50,Reset: IN std_logic;
DataRGB: OUT std_logic_vector( 29 downto 0);
VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS: OUT std_logic;
CE:OUT std_logic;
OE:BUFFER std_logic;
WE: OUT std_logic;
LB,UB: BUFFER std_logic;
AddressRam: OUT std_logic_vector(17 downto 0);
DataRam: INOUT std_logic_vector( 15 downto 0);
DataRXOUT: OUT std_logic_vector(7 downto 0));
end component;
signal Tx_Line,Start,Flag2:  std_logic;
signal Clock_50,Clock_27,Reset:  std_logic;
signal DataRGB:  std_logic_vector( 29 downto 0);
signal VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS: std_logic;
signal CE: std_logic;
signal OE: std_logic;
signal WE:  std_logic;
signal LB,UB:  std_logic;
signal AddressRam:  std_logic_vector(17 downto 0);
signal DataRam,Dato: std_logic_vector( 15 downto 0);
BEGIN
PROCESS
```

```
BEGIN
Reset<='1';
Tx_LINE<='1';
Start<='0';
wait for 30 ns;
Reset<='0';
Start<='1';
wait for 40 ns;
wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO W
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO YH
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='0';      --Bit 0
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO YL
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='0';      --Bit 0
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
```

```
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO XH
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='0';      --Bit 0
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO XL
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='0';      --Bit 0
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='0';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO R
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
```

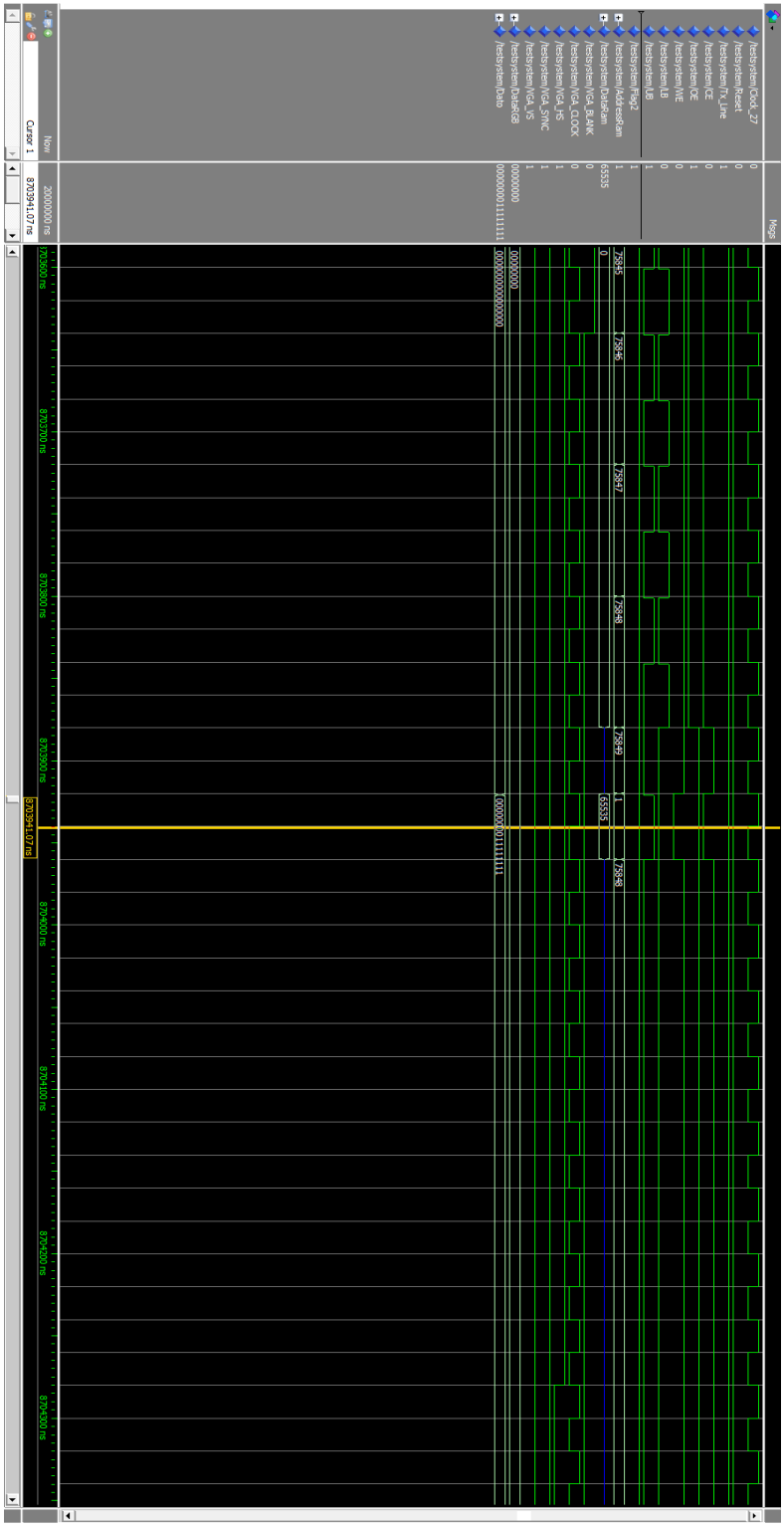
```
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO G
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
--wait for 416640 ns; --MANDiAMO 4 dati 1
---MANDO B
Tx_line<='0';      --Start
wait for 104160 ns;
Tx_line<='1';      --Bit 0
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';
wait for 104160 ns;
Tx_line<='1';      --Bit Stop
wait for 104160 ns;
end process;
PROCESS
BEGIN
clock_50<='1';
wait for 10 ns;
clock_50<='0';
wait for 10 ns;
end process;
PROCESS
BEGIN
clock_27<='1';
```

```
wait for 18.18 ns;
clock_27<='0';
wait for 18.18 ns;
end process;
PRocess(RESET,CE,OE,WE,UB,LB,AddressRam,DataRam,Dato)
BEGIN
if(Reset='1')then
DATo<=(others=>'0');
elsif(CE='0' AND OE='0' AND WE='1')then
if(AddressRam="000000000000000000")then
DataRam<=Dato;
else
DataRam<=(others=>'0');
end if;
elsif(CE='0' AND OE='1' AND WE='0')then
if(AddressRam="000000000000000000" AND UB='0' AND LB='1')then
Dato(15 downto 8)<=DataRam(15 downto 8);
elsif(AddressRam="000000000000000000" AND UB='1' AND LB='0')then
Dato(7 downto 0)<=DataRam(7 downto 0);
end if;
else
DataRam<=(others=>'Z');
end if;
end process;
SYS: SYstem PORT MAP(
Start,Flag2,Tx_Line,Clock_27,Clock_50,Reset,DataRGB,VGA_BLANK,VGA_SYNC,VGA_CLOCK,VGA_HS,VGA_VS,
CE,OE,WE,LB,UB,AddressRam,DataRam);
end architecture;
```

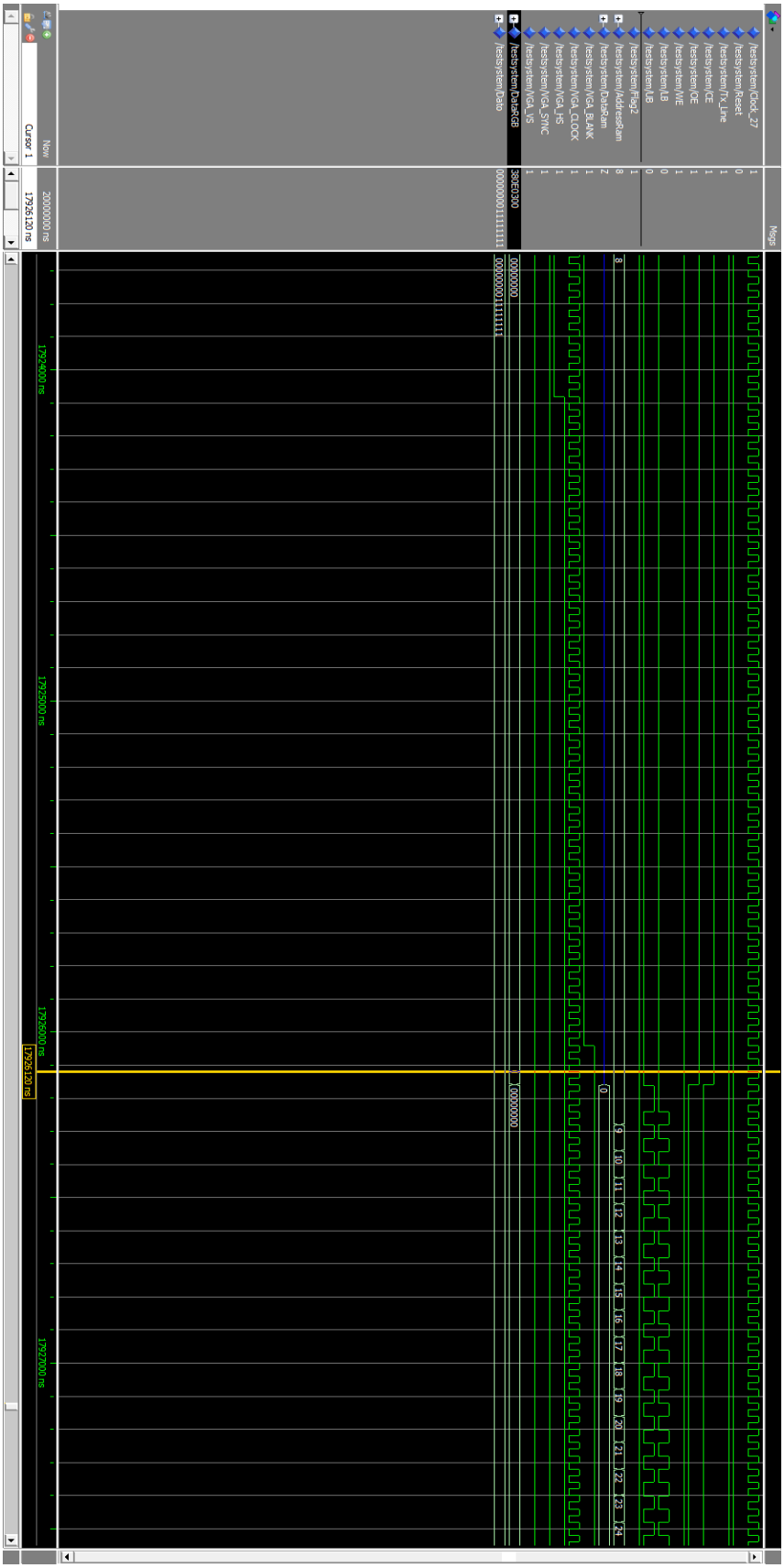
Timing

Di seguito è mostrato il momento in cui avviene la scrittura del pixel in memoria ed il momento in cui, nel secondo ciclo di refresh, appare in uscita il dato sulla porta DataRGB connessa al DAC subito dopo il Vertical Sync back porch.

Scrittura Pixel in Ram



Rappresentazione del Pixel nel secondo ciclo di Refresh



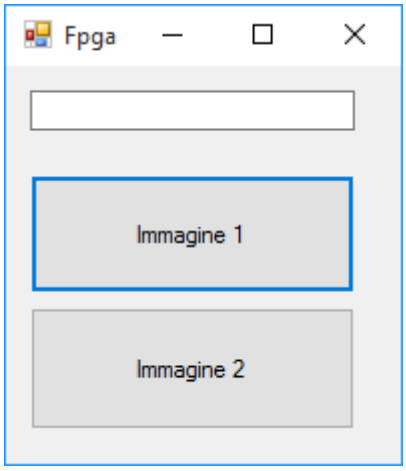
5.2 Test con la scheda DE2 Board

Dopo aver assegnato correttamente i pin del FPGA è stato verificato il funzionamento del sistema. Per inviare i dati sulla linea seriale è stata creata un applicazione C# in grado di trasferire due possibili immagini.

5.2.1 Applicazione C#

L'applicazione presenta una textBox e due tasti: "Immagine 1", "Immagine2". Dopo la pressione di un tasto viene mostrato il messaggio "Start Immagine n". Al termine dell'invio viene mostrato il messaggio "End".

Screenshoot



Codice C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.IO;
using System.IO.Ports;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      SerialPort serialPort1;
      public Form1() {
         InitializeComponent();
      }
      private void button1_Click(object sender, EventArgs e)
      {
         serialPort1 = new System.IO.Ports.SerialPort("COM3", 9600);
         serialPort1.Open();
         int y,x;
         byte b;
```

```
        textBox1.Text = "Start Immagine 1";
        for ( y = 0; y < 480; y++){
        for( x = 0; x < 640; x++)
        {
          serialPort1.Write("W");
          b = Convert.ToByte(y >> 8);
          serialPort1.Write(new byte[] { b }, 0, 1);
          b = Convert.ToByte( y & 255);
          serialPort1.Write(new byte[] { b }, 0, 1);
          b = Convert.ToByte(x >> 8);
          serialPort1.Write(new byte[] { b }, 0, 1);
          b = Convert.ToByte(x & 255);
          serialPort1.Write(new byte[] { b }, 0, 1);
          serialPort1.Write(new byte[] { 1 }, 0, 1);
          serialPort1.Write(new byte[] { 115 }, 0, 1);
          serialPort1.Write(new byte[] { 255 }, 0, 1);

          }
        }
        textBox1.Text = "End";
        serialPort1.Close();
      }
      private void Form1_Load(object sender, EventArgs e) {
      }
      private void button2_Click(object sender, EventArgs e)
      {
        serialPort1 = new System.IO.Ports.SerialPort("COM3", 9600);
        serialPort1.Open();
        textBox1.Text = "Start Immagine 2";
        int y, x;
        byte b;
        for (y = 0; y < 480; y++)
        {
          for (x = 0; x < 640; x++)
          {
            serialPort1.Write("W");
            b = Convert.ToByte(y >> 8);
            serialPort1.Write(new byte[] { b }, 0, 1);
            b = Convert.ToByte(y & 255);
            serialPort1.Write(new byte[] { b }, 0, 1);
            b = Convert.ToByte(x >> 8);
            serialPort1.Write(new byte[] { b }, 0, 1);
            b = Convert.ToByte(x & 255);
            serialPort1.Write(new byte[] { b }, 0, 1);
            b = Convert.ToByte(x & 255);
            serialPort1.Write(new byte[] { b }, 0, 1);
            serialPort1.Write(new byte[] { b }, 0, 1);
            serialPort1.Write(new byte[] { b }, 0, 1);
          }
        }
       textBox1.Text = "End";
        serialPort1.Close();
      }
    }
}
```

5.2.2 Test

Di seguito sono riportate le immagini trasferite al sistema. La prima durante dove sono rappresentate delle strisce verticali e la seconda di colore arancione. L'immagine rappresentata nella parte inferiore dello schermo è il contenuto, casuale, della SRAM non ancora riempita. L'immagine arancione è stata inviata fermando il trasferimento della prima. Le schermate non sono complete in quanto il trasferimento completo richiede molto tempo.

Immagine 1

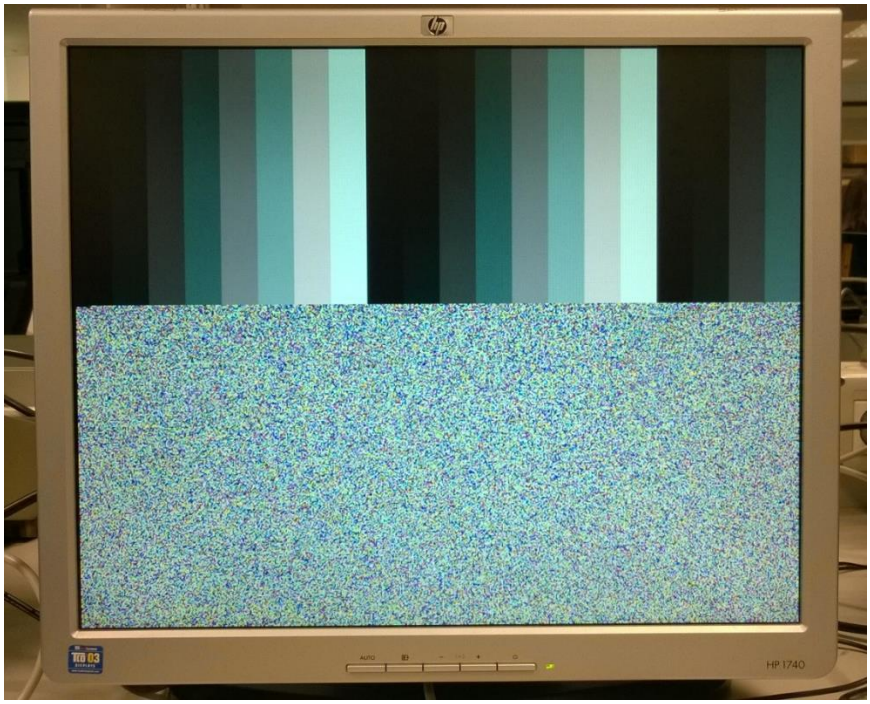


Immagine 2