ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Mastermind on FPGA: logic and implementation.

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria Informatica
Progetto di Sistemi Digitali M - Prof. Eugenio Faldella

Davide Di Donato Marco Valli Silvia Damiani

Anno Accademico 2017/2018

Sommario

Lo scopo del progetto è simulare la versione del gioco da tavolo Mastermind, utilizzando il linguaggio VHDL, realizzando un'architettura su FPGA che permetta all'utente di giocare in tempo reale: in particolare permetterà al decifratore, una volta codificata la sequenza, di inserire le combinazioni di colori desiderate e visualizzarne la correttezza a video secondo una logica predefinita, espressa prima dell'inizio della partita.

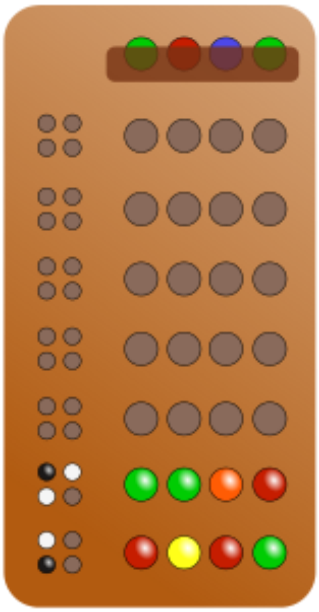
Lo studio e l'implementazione si basano sulle regole del gioco da tavolo e sugli algoritmi per la generazione di giochi digitali.

1 Introduzione

1.1 Introduzione a MASTERMIND

Il Mastermind è un gioco, nato su carta e penna, sviluppatosi come gioco da tavolo e oggi implementato su dispositivi digitali.

Mastermind si compone di due giocatori: il codificatore della sequenza e il decifratore. Il primo genera inizialmente una sequenza di colori ed è obiettivo del secondo riuscire a decodificare la sequenza in un numero prefissato di tentativi. Per creare la versione digitale di questo gioco strategico matematico sono state apportate alcune modifiche che osserveremo di seguito.



I Mastermind gioco da tavolo

1.2 Versione originale

Nella versione originale il gioco è composto di una scheda di decodifica con pioli colorati, bianchi e neri; il codificatore memorizza la sequenza di colori in una sezione nascosta, il decifratore pone le ipotesi e in seguito il generatore fornisce un feedback per ogni ipotesi:

- Il numero di colori esatti al posto esatto, con pioli neri,
- Il numero di colori esatti al posto sbagliato, con pioli bianchi,

non comunicando quali colori siano esatti e quali errati ma solo quanti. È possibile utilizzare anche più volte lo stesso colore all'interno della sequenza, composta di quattro colori, combinazione delle sei colorazioni disponibili. Se il decodificatore riesce a indovinare la sequenza entro il

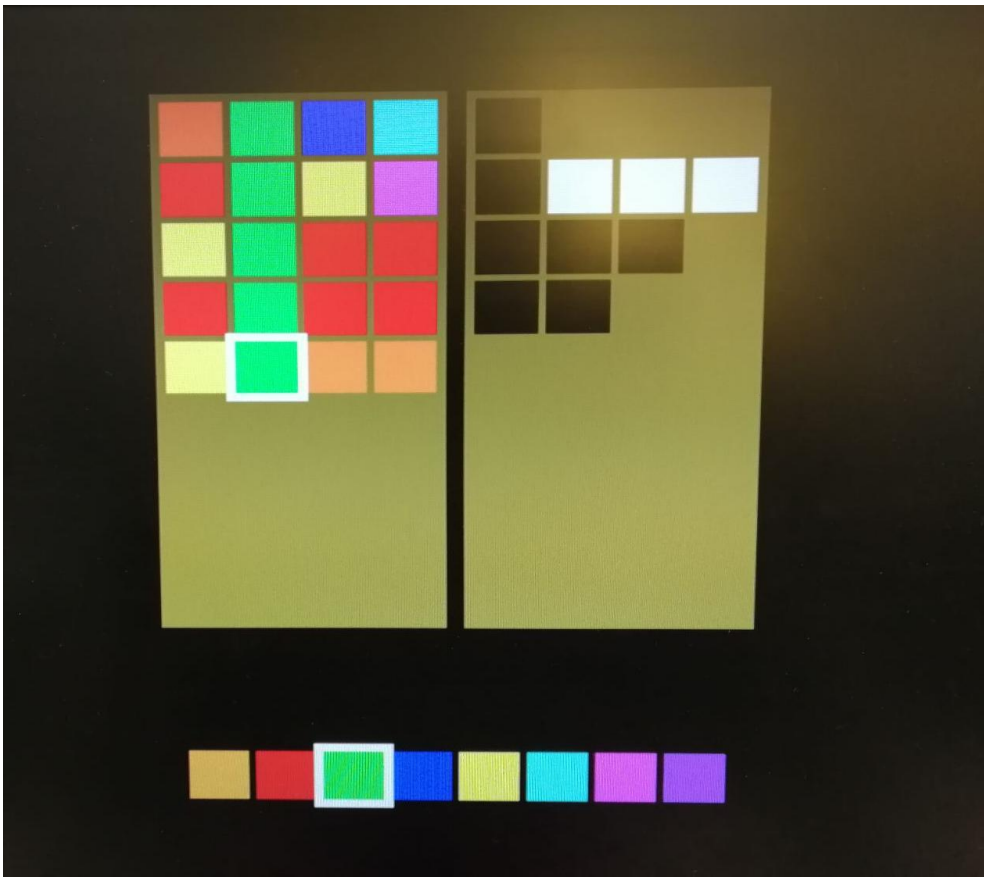
numero di tentativi predeterminati (solitamente i tentativi sono 9) allora quest'ultimo vince la partita, altrimenti vince il codificatore.

1.3 Versione digitale

Nella versione digitale la parte del codificatore è affidata al software e si ha quindi un singolo giocatore. Anche in questo caso si utilizza una sequenza di quattro colori ma combinazione di ben otto colori differenti; viene fornita la possibilità di utilizzare più volte lo stesso colore e i feedback ricevuti sono identici a quelli della versione analogica, con pioli neri e bianchi per indicare il numero di colori esatti in posti giusti o errati.

In questa versione la scheda di decodifica è riportata sullo schermo LCD collegato alla FPGA, la sezione nascosta presente nel gioco da tavolo è assente nella versione digitale; tuttavia, sia in caso di vittoria sia di sconfitta, la corretta sequenza apparirà nella schermata finale. Tutto ciò che nel gioco fisico è indicato con pioli colorati di piccole e grandi dimensioni, è indicato sullo schermo con quadrati di ugual dimensione e diversa colorazione.

Il numero di tentativi resta invariato a nove.



II Esempio Mastermind in formato digitale

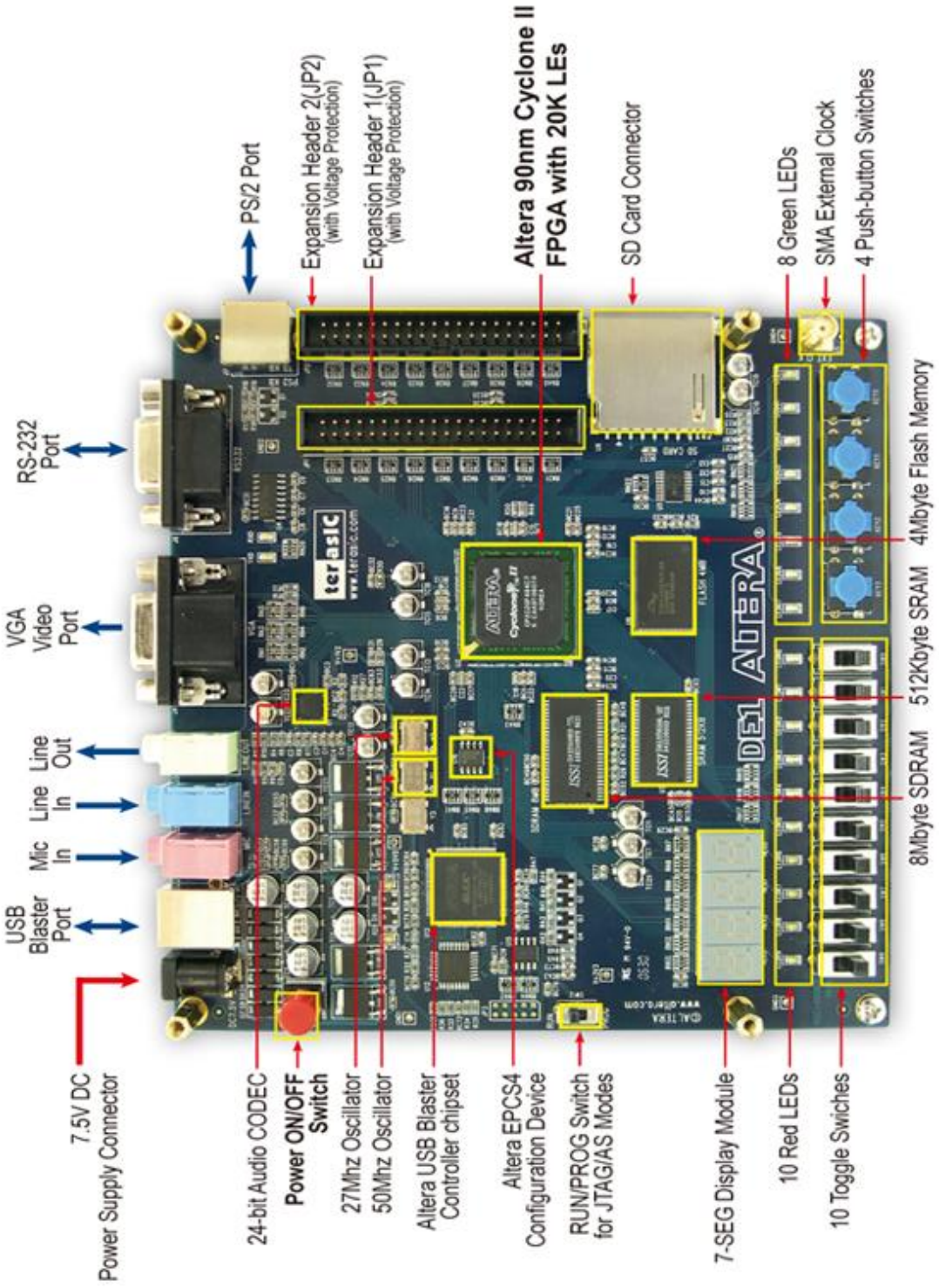
1.4 Strumenti di sviluppo utilizzati

Il progetto è stato testato sulla scheda ALTERA DE1 dotata di risorse logiche e memoria sufficienti all'implementazione del gioco. La maggior parte dei componenti sono evidenziati in figura, nella pagina successiva, riportiamo quindi solo i principali:

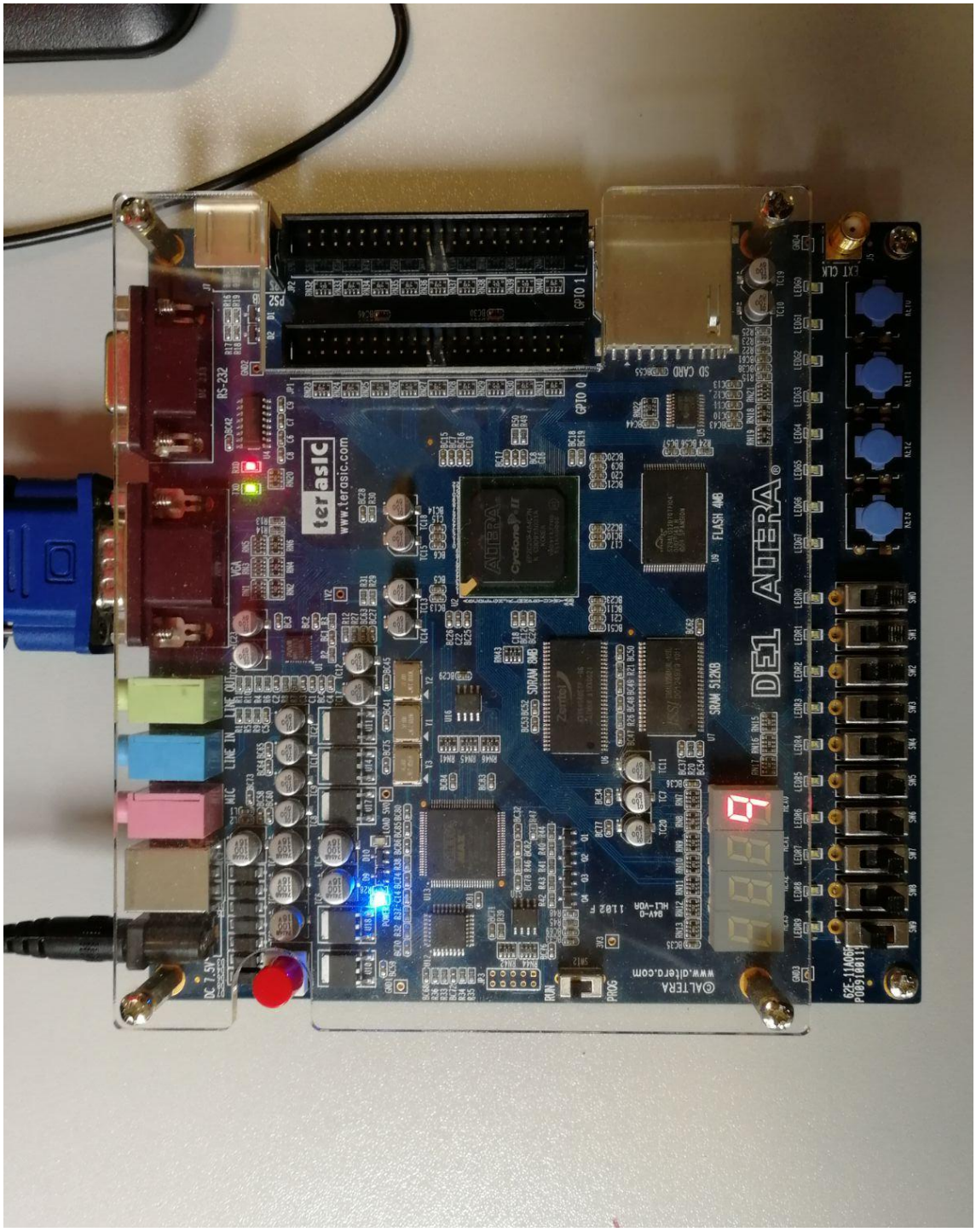
- Cyclone II FPGA: è il cuore della scheda, contenente 20000 risorse logiche.
- Dispositivo di configurazione seriale EPCS4 utilizzata per consentire la programmazione dall'esterno
- 8Mbyte SDRAM, 4Mbyte FLASH, 512 Kbyte SRAM
- VGA: utilizzata per visualizzare il video su monitor VGA.
- Display a 7 segmenti: countdown utilizzato per visualizzare il numero di tentativi rimasti
- Pulsanti: permettono al giocatore di inviare comandi, quindi di poter giocare, utilizzando come sorgente il video.
- LED e switch: utilizzati per varie funzionalità come testing, feedback e reset.

Oltre alla scheda sono stati utilizzati:

- Monitor LCD



III Scheda Altera DE1 e componenti principali



IV Setup Board

2 Descrizione dell'architettura

L'architettura generale si basa sul pattern MVC:

- **model**, è il nostro *datapath* che gestisce l'insieme dei dati, le modifiche e risponde alle singole interrogazioni sui dati;
- **view**, gestisce l'area di visualizzazione sullo schermo LCD che rende visibile all'utente i dati gestiti dal model;
- **controller**, la *control Unit* gestisce i singoli input dell'utente e invia i comandi rispettivamente al model e/o alla view a seconda delle operazioni richieste.

Inizialmente il sistema è acceso e rimane in attesa della disattivazione del reset iniziale da parte del singolo giocatore (switch 9); una volta che la control unit riceve il segnale lo trasmette alla view che proietta il nome del gioco e le indicazioni per iniziare una nuova partita. Una volta premuto uno dei bottoni di START, la control unit invia il segnale al model il quale genererà la sequenza da indovinare, mentre a livello view uno schema identico viene ripetuto riga dopo riga fino al termine della partita.

Quattro quadrati consentono di inserire e modificare la sequenza di colori utilizzando i bottoni presenti sulla scheda FPGA; la gestione di questi segnali è lasciata alla control unit che in base all'input invia alla view di ridisegnare la scena e al datapath le variazioni da apportare al modello in base ai pulsanti premuti dall'utente. Ulteriori quadrati, nella seconda parte dello schermo, sono utilizzati per il feedback sulla sequenza inserita: una volta confermata la sequenza, il datapath elabora i singoli valori, confronta la sequenza inserita in input con quella segreta memorizzata inizialmente e restituisce il risultato dell'elaborazione alla view per poter essere osservato dall'utente. Le sequenze precedenti non possono andare perse bensì vanno salvate, in particolare: la scena da visualizzare sullo schermo viene temporaneamente salvata nella SRAM della DE1, utilizzata come buffer, mentre i tentativi inseriti dall'utente e i corrispettivi feedback restituiti dal gioco sono memorizzati nei blocchi logici dell'FPGA.

L'algoritmo di gioco, così come i principali componenti per lo sviluppo e la grafica, sono implementati in macro blocchi specifici; alcune entità come quelle per la generazione di numeri random, la definizione di costanti o tipi e la gestione della view saranno discusse in seguito, per ora ci limitiamo ad elencare i principali:

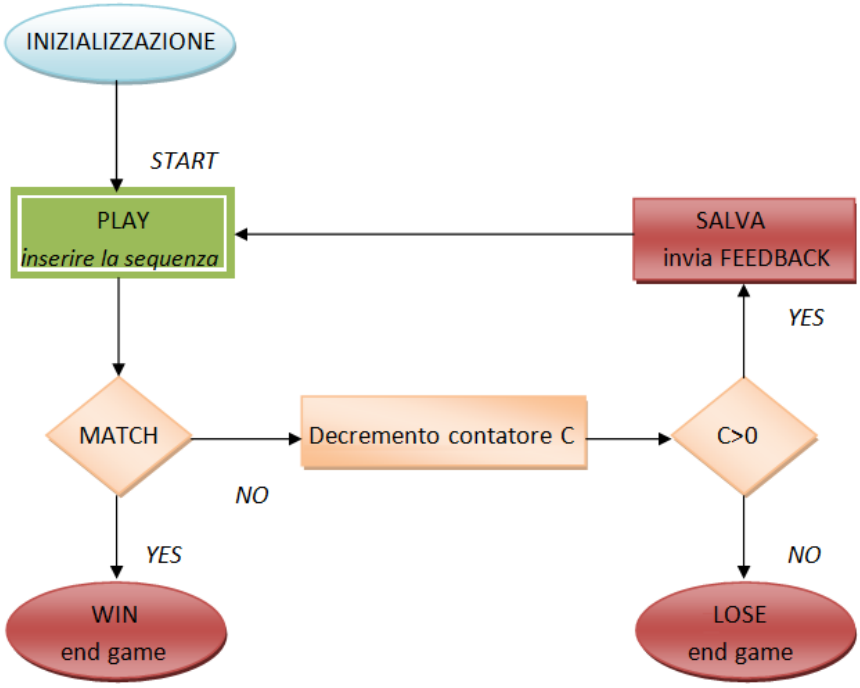
- CONTROLLER: definisce una serie di segnali di input e output, azioni a cui corrispondono modifiche su datapath e view;
- DATAPATH: al suo interno sono definiti gli stati della macchina, viene generata la sequenza segreta, gestito il reset, effettuate le operazioni di elaborazione come il match fra le sequenze e salvate combinazioni e feedback;
- VGA_*: serie di blocchi per la gestione della view;
- TEXT_*: file per la gestione del testo, correlati ai blocchi VGA;
- MASTEMIND: main principale dell'intero progetto, si occupa dell'identificazione dei segnali e della loro gestione a livello di clock, controller, datapath e view.

Sulla scheda è implementato uno switch per il reset del gioco una volta completato o nel caso in cui si voglia semplicemente giocare una nuova partita pur non avendo terminato la precedente.

Altro feedback è fornito dal display a 7 segmenti in cui si osservano i tentativi restanti e simboli in caso di sconfitta o vittoria.

2.1 Diagramma di flusso

Di seguito la sequenza di operazioni alla base dell'algoritmo di gioco, in azzurro la fase di inizializzazione, in verde l'input, in arancione le fasi di elaborazione e in rosso gli output.



Anche se non indicato dalle frecce, in seguito all'end game tramite reset forzato si torna all'inizializzazione.

Gli stati evidenziati sono quattro ma a livello implementativo ne abbiamo inserito un quinto come "utility", uno stato INITIAL successivo all'inizializzazione e allo start in cui è generata la combinazione segreta; subito dopo si entra nello stato play di gioco e s'inizia la partita.

3 Implementazione

Di seguito riportiamo i vari blocchi d'implementazione seguendo uno schema logico temporale: inizialmente sono stati implementati il base_package con le costanti e i tipi di base utilizzati lungo il progetto, poi l'interfacciamento con la SRAM e con la VGA; successivamente, una volta implementati controller e datapath grazie ad una serie di piccoli blocchi di supporto per la generazione di singoli componenti, il blocco mastermind è stato generato.

L'idea alla base è di utilizzare un'architettura in cui Mastermind è l'entità top-level e si occupa di associare i componenti hardware della scheda alle entità a essa sottostanti (model, datapath, …), in particolare, prima vengono instanziare le entità e i segnali e successivamente sono espresse le modalità con cui tali segnali vengono utilizzati per collegare le entità tra loro.

Codice delle principali costanti e dei tipi in uso nel progetto.

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use work.vga_package.all;

package base_package is

	-- Numero di tentativi possibili
	constant GUESSES_NUMBER : positive := 9;

	-- Numero di colori che l'utente deve impostare per
ogni tentativo
	constant SQUARES_NUMBER : positive := 4;

	-- Numero di colori diversi tra cui l'utente può
scegliere
	constant COLORS_NUMBER  : positive := 8;

	-- Numero di tipi di indizi che il computer può dare
all'utente
	-- Tipo 0 -> colore giusto e posto giusto
	-- Tipo 1 -> colore giusto e posto sbagliato
	-- Tipo 2 -> colore sbagliato
	constant HINTS_NUMBER   : positive := 3;

	-- Tipo che descrive, per ogni colore da impostare per
tentativo, che colore può essere
	-- Ognuno di questi colori è definito nel file
vga_package
	subtype guess_peg   is natural range 0 to
(COLORS_NUMBER - 1);
	-- Tipo che descrive il tentativo dell'utente, che è
un'array di 4 colori che l'utente deve impostare
	type guess       is array(0 to (SQUARES_NUMBER - 1)) of
guess_peg;
	-- Tipo che descrive l'insieme dei tentativi
effettuati, quindi un'array di 9 tentativi (guess)
	type guess_board is array(0 to (GUESSES_NUMBER - 1)) of
guess;

	-- Tipo che descrive il singolo colore dell'indizio,
che può essere:
	-- Bianco per colore giusto posto sbagliato
	-- Nero per colore giusto posto giusto
	-- Marrone quando il colore non e' giusto
	type hint_peg       is (PLACE, COLOR, NOTHING);
	-- Tipo che descrive l'indizio dato dal computer
	type hint        is array(0 to (SQUARES_NUMBER - 1)) of
hint_peg;
	-- Tipo che descrive l'insieme degli indizi dati dal
computer, quindi un'array di 9 indizi (hint)
	type hint_board  is array(0 to (GUESSES_NUMBER - 1)) of
hint;

end package;
```

3.1 Generazione della sequenza random

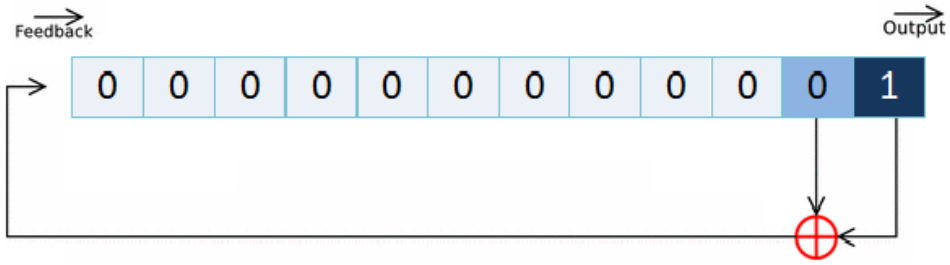
Alla base del Mastermind vi è la generazione di una sequenza puramente casuale di colori, nel caso specifico bit, da indovinare per vincere la partita.

Il progetto fa uso di un *registro a scorrimento a retroazione lineare (LFSR)* per la generazione casuale in output della sequenza; i dati in ingresso sono prodotti da una funzione lineare dello stato interno, parliamo di funzioni lineari a singoli bit e quindi i possibili bit d'ingresso sono prodotti dall'or esclusivo (xor) di alcuni bit memorizzati all'interno dei registri. Il seme iniziale rende l'operazione del registro deterministica, la sequenza di valori è determinata dallo stato corrente e da quello precedente, il numero finito di stati possibili implica la ripetizione dei valori in uscita ma, se viene scelta una funziona di retroazione ben strutturata, la sequenza di bit appare casuale e ha un periodo particolarmente lungo, se non altro adatto alle nostre tempistiche di gioco.

3.1.1 Funzionamento

LFSR massimale: produce una *sequenza-n*, ossia passa attraverso tutti gli stati del registro di traslazione, salvo che lo stato iniziale non sia composto di soli zeri e in tal caso l'uscita rimane costante.

La sequenza di numeri prodotta da un LFSR può essere considerata un sistema numerico binario valido quanto il codice Gray o il codice binario naturale. La sequenza di **tap**, ossia di uscite che influenzano l'ingresso di un LFSR, può essere rappresentata come un polinomio modulo 2: parliamo del polinomio caratteristico (polinomio di retroazione) in cui i coefficienti del polinomio devono essere 1 o 0.



V LFSR

Nel nostro caso i tap sono all'12° e al 11° bit e il relativo polinomio si costruisce con la somma delle potenze dei termini dei bit considerati *tap* **+1** (che non corrisponde ad un tap)

$$x^{11} + x^{10} + 1$$

La scelta è stata fatta nel rispetto delle seguenti regole e ricordando che possono esserci più sequenze di tap che rendono massimale un LFSR:

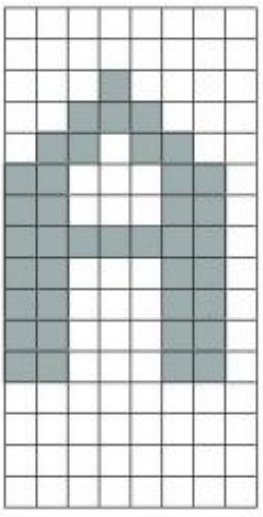
- Se e solo se il polinomio è primitivo, allora LFSR è massimale
- LFSR è massimale se e solo se il numero di tap è pari
- I valori dei tap saranno primi tra loro (MCD=1)

Di seguito il codice:

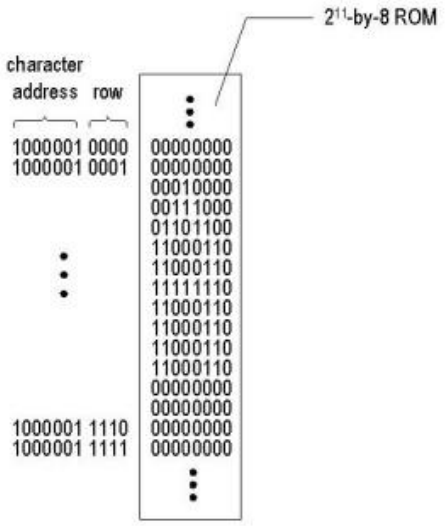
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity randomgen is
  Port (
   clk:in std_logic;
   a:out std_logic_vector(11 downto 0));
end randomgen;

architecture Behavioral of randomgen is

begin

    Gen:process is
    variable temp:std_logic_vector(11 downto 0) :=
"000000000001";
    begin
        temp := temp( 10 downto 0 )  & ( temp(11) xor
temp(10) );
        a <= temp;
        wait until (clk = '0');
    end process Gen;

end Behavioral;
```

3.2 Generazione del testo

Il primo problema da affrontare a livello grafico è come rappresentare simboli e caratteri a video; essendo un linguaggio hardware, VHDL non implementa nessun tipo di libreria per proiezione di caratteri: immaginiamo quindi di avere a disposizione delle tessere, i modelli in ogni tessera costituiscono il set di caratteri.



VI Modello Pixel



VII Contenuto del Font Rom

Utilizziamo un font 8x16 (come i primi PC IBM) quindi ogni carattere è rappresentato con un pattern di 8x16 pixel. I pattern per i caratteri sono memorizzati in una memoria apposita nota come FONT_ROM.

Di seguito riportiamo il codice e alcuni esempi di composizione di caratteri ricordando che è possibile creare migliaia di combinazioni ASCII e altre forme geometriche ben definite.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.text_package.all;


entity font_rom is
      port(
            clk:  in std_logic;
            addr: in integer;
            font_row:    out    std_logic_vector(FONT_WIDTH-1
downto 0)
      );
end font_rom;

architecture behavioral of font_rom is

      type   rom_type   is   array   (0   to   2**11-1)   of
std_logic_vector(FONT_WIDTH-1 downto 0);

      signal ROM: rom_type := (

            -- code x12
            "00000000", -- 0
            "00000000", -- 1
            "00011000", -- 2     **
            "00111100", -- 3    ****
            "01111110", -- 4   ******
            "00011000", -- 5     **
            "00011000", -- 6     **
            "00011000", -- 7     **
            "01111110", -- 8   ******
            "00111100", -- 9    ****
            "00011000", -- a     **
            "00000000", -- b
            "00000000", -- c
            "00000000", -- d
            "00000000", -- e
            "00000000", -- f

            -- A: code x41
            "00000000", -- 0
            "00000000", -- 1
            "00010000", -- 2     *
            "00111000", -- 3    ***
            "01101100", -- 4   ** **
            "11000110", -- 5 **   **
            "11000110", -- 6 **   **
            "11111110", -- 7 *******
            "11000110", -- 8 **   **
            "11000110", -- 9 **   **
            "11000110", -- a **   **
            "11000110", -- b **   **
            "00000000", -- c
            "00000000", -- d
            "00000000", -- e
            "00000000", -- f
```

```
-- B: code x42
            "00000000", -- 0
            "00000000", -- 1
            "11111100", -- 2 ******
            "01100110", -- 3  **  **
            "01100110", -- 4  **  **
            "01100110", -- 5  **  **
            "01111100", -- 6  *****
            "01100110", -- 7  **  **
            "01100110", -- 8  **  **
            "01100110", -- 9  **  **
            "01100110", -- a  **  **
            "11111100", -- b ******
            "00000000", -- c
            "00000000", -- d
            "00000000", -- e
            "00000000", -- f

      );
begin

      pixelOn: process (clk)
      begin
            if rising_edge(clk) then
                  font_row <= ROM(addr);
            end if;
      end process;

end behavioral;
```

3.3 Comunicazione con l'utente: gestione del testo

In un gioco è importante che l'utente ottenga sempre indicazioni sulle azioni da compiere e feedback su di esse; in questo caso è fondamentale comunicare all'utente il nome del gioco, come avviare una partita, la vittoria o la sconfitta.

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

package text_package is

 constant FONT_WIDTH : integer := 8;
 constant FONT_HEIGHT : integer := 16;

 constant TEXT_X : integer := 512/2;
 constant TEXT_Y : integer := 480/2;

 constant TITLE_TEXT : string := "M A S T E R M I N D";
 constant INTRO_TEXT : string := "Premi un qualsiasi KEY da 1
a 3 per iniziare a giocare";
 constant WIN_TEXT : string := "Hai vinto! La combinazione e'
corretta!";
 constant LOSE_TEXT : string := "Hai perso! La combinazione
era:";

 type message is (TITLE, INTRO, WIN, LOSE);

 type codes is array(natural range<>) of integer;

end package;
```

I testi sono gestiti da un apposito controller che si comporta da arbitro e coordina i diversi feedback in base ai segnali ricevuti e agli eventi in corso; riceve in ingresso il clock, il testo da considerare nell'array di testi predefinito (selezionabili uno alla volta), next_bit e next_line che indicano rispettivamente, dato un carattere, il valore successivo rispetto al bit attuale lungo la stessa linea e la linea successiva; in output otterremo un bit che indica se il pixel si trova all'interno del carattere o meno, quindi se il bit è impostato ad 1 o a 0.

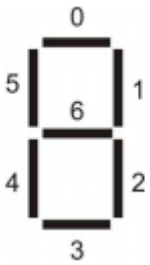
Indipendentemente dal tipo di testo da proiettare, l'entità font_rom ci permette di ottenere l'array di bit che costituisce la riga attuale del carattere corrente, mentre l'indirizzamento è coordinato da 3 integer rispettivamente per i singoli bit, per i distinti caratteri e per le righe: si scorre di bit in bit per ogni carattere fino al termine della linea con un semplice algoritmo composto da istruzioni condizionali e contatori.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.text_package.ALL;

entity text_controller is
 port (
  CLOCK     : in std_logic;
  RESET_N     : in std_logic;
  CHOSEN_TEXT   : in message;
  NEXT_BIT      : in std_logic;
  NEXT_LINE     : in std_logic;
  PIXEL     : out std_logic := '0'
 );
end text_controller;


architecture behavioral of text_controller is

 signal y_counter    : integer;
 signal font_address : integer;
 signal char_bit_in_row: std_logic_vector(FONT_WIDTH-1 downto
0) := (others => '0');
 signal char_position:integer := 0;
 signal bit_position:integer := 0;
 signal char_codes   : codes(0 to message'POS(message'HIGH));
begin

 font_rom: entity work.font_rom
  port map(
   clk  => CLOCK,
   addr => font_address,
   font_row => char_bit_in_row
 );

 process(CLOCK, RESET_N)
  variable last_chosen_text: message := TITLE;
 begin
  if RESET_N = '0' or last_chosen_text /= CHOSEN_TEXT then
   y_counter      <= 0;
   char_position <= 1;
   bit_position  <= 0;
   last_chosen_text := CHOSEN_TEXT;
  elsif rising_edge(CLOCK) then
   if NEXT_BIT = '1' then
    if bit_position = 7 then
     bit_position <= 0;
     char_position <= char_position + 1;
    else
     bit_position <= bit_position + 1;
    end if;
   end if;
   if NEXT_LINE = '1' then
    bit_position <= 0;
    char_position <= 1;
    y_counter <= y_counter + 1;
   end if;
  end if;
 end process;

 char_codes(0) <= character'pos(TITLE_TEXT(char_position));
```

```
 char_codes(1) <= character'pos(INTRO_TEXT(char_position));
 char_codes(2) <= character'pos(WIN_TEXT(char_position));
 char_codes(3) <= character'pos(LOSE_TEXT(char_position));

 font_address <= char_codes(message'POS(CHOSEN_TEXT)) * 16 +
y_counter;

 pixel <= char_bit_in_row(FONT_WIDTH-bit_position);

end behavioral;
```

3.4 Display 7 segmenti

Un display a 7 segmenti è un dispositivo elettronico in grado di visualizzare cifre o lettere attraverso l'accensione, con diverse combinazioni, di 7 segmenti luminosi; il pilotaggio è permesso da un dispositivo integrato con funzione di decodifica BCD (binary-coded decimal) a "7 segmenti". La scheda DE1 ha quattro display a 7 segmenti per poter mostrare numeri fino alle migliaia; i sette segmenti sono collegati ai pin sull'FPGA, a livello logico basso il segmento si accende mentre l'applicazione di un livello logico alto lo spegne. Ogni segmento è identificato da un indice da zero a sei.

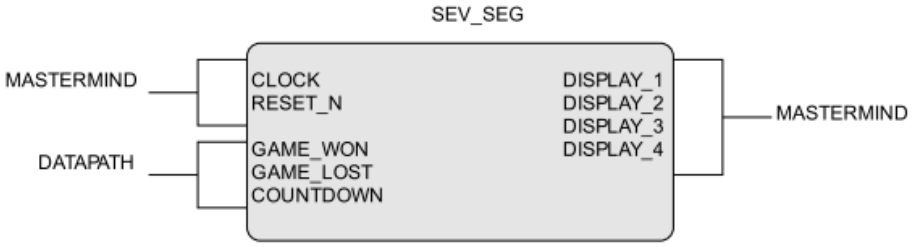


VIII Posizione e indice di ogni segmento

Il display è stato utilizzato per due differenti feedback.

- conto alla rovescia dei tentativi rimanenti: per il conto alla rovescia è stato utilizzato un unico display, al reset il display risulta spento, all'avvio della partita viene settato a 9 e tentativo dopo tentativo viene decrementato fino al valore 0 in caso di perdita.

- Vittoria o sconfitta della partita: nel primo caso, in tutti e quattro i display, a partire dal segmento in alto (0) e seguendo il senso orario si accendono uno alla volta tutti i led escluso quello centrale (6) a formare un cerchio rotante; nel secondo caso, sempre su tutti i display si accende una X lampeggiante, si illuminano quindi il led centrale e quelli laterali e restano spenti quello inferiore e superiore (0,3).



IX Interfaccia Display 7 segmenti

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.base_package.all;


entity sev_seg is
 port
 (
  CLOCK             : in std_logic;
  RESET_N           : in std_logic;

  GAME_WON          : in std_logic;
  GAME_LOST         : in std_logic;
  COUNTDOWN         : in natural;

  DISPLAY_1         : out std_logic_vector(6 downto 0);
  DISPLAY_2         : out std_logic_vector(6 downto 0);
  DISPLAY_3         : out std_logic_vector(6 downto 0);
  DISPLAY_4         : out std_logic_vector(6 downto 0)
 );
end sev_seg;

architecture behavioral of sev_seg is

 constant clockwise_circle_phases : natural := 6;
 constant flashing_x_phases       : natural := 2;

 signal animation_clock   : std_logic;
 signal current_phase     : natural;
 signal current_animation : std_logic_vector(6 downto 0);

 function clockwise_circle(x : natural)
  return std_logic_vector is
 begin
  case(x) is
   when 0      => return "1011111";
   when 1      => return "1101111";
   when 2      => return "1110111";
   when 3      => return "1111011";
   when 4      => return "1111101";
   when 5      => return "1111110";
   when others => return "1111111";
  end case;
 end function;

 function flashing_x(x : natural)
  return std_logic_vector is
 begin
  case(x) is
   when 0      => return "0001001";
   when 1      => return "1111111";
   when others => return "1111111";
  end case;
 end function;

begin

 display: process (CLOCK, RESET_N) is
   begin
  if (RESET_N = '0') then
```

```
  current_animation <= "1111111";
  current_phase <= 0;
  DISPLAY_1 <= "1111111";
  DISPLAY_2 <= "1111111";
  DISPLAY_3 <= "1111111";
  DISPLAY_4 <= "1111111";

 elsif (rising_edge(CLOCK)) then
  if (GAME_WON = '1' and animation_clock = '1') then
   current_animation <= clockwise_circle(current_phase);
   DISPLAY_1 <= current_animation;
   DISPLAY_2 <= current_animation;
   DISPLAY_3 <= current_animation;
   DISPLAY_4 <= current_animation;
   if current_phase + 1 = clockwise_circle_phases then
    current_phase <= 0;
   else
    current_phase <= current_phase + 1;
   end if;

  elsif (GAME_LOST = '1' and animation_clock = '1') then
   current_animation <= flashing_x(current_phase);
   DISPLAY_1 <= current_animation;
   DISPLAY_2 <= current_animation;
   DISPLAY_3 <= current_animation;
   DISPLAY_4 <= current_animation;
   if current_phase + 1 = clockwise_circle_phases then
    current_phase <= 0;
   else
    current_phase <= current_phase + 1;
   end if;

  else
  -- qui il display 1 mostra il countdown
   if (GAME_WON = '0' and GAME_LOST = '0') then
    case (COUNTDOWN) is
     when 1 =>    DISPLAY_1 <= "1111001";
     when 2 =>      DISPLAY_1 <= "0100100";
     when 3 =>    DISPLAY_1 <= "0110000";
     when 4 =>    DISPLAY_1 <= "0011001";
     when 5 =>    DISPLAY_1 <= "0010010";
     when 6 =>    DISPLAY_1 <= "0000010";
     when 7 =>    DISPLAY_1 <= "1111000";
     when 8 =>    DISPLAY_1 <= "0000000";
     when 9 =>  DISPLAY_1 <= "0011000";
     when others => DISPLAY_1 <= "1111111";
    end case;
   end if;
  end if;
 end if;
end process display;

animation_clock_gen : process(CLOCK, RESET_N) is
 variable counter : natural range 0 to (5000000 - 1);
begin
 if (RESET_N = '0') then
  animation_clock <= '0';
  counter := 0;

 elsif (rising_edge(CLOCK)) then
  if counter = counter'HIGH then
```

```
      counter := 0;
      animation_clock <= '1';
     else
      counter := counter + 1;
      animation_clock <= '0';
     end if;
    end if;
  end process ;

end behavioral;
```

3.5 Controller

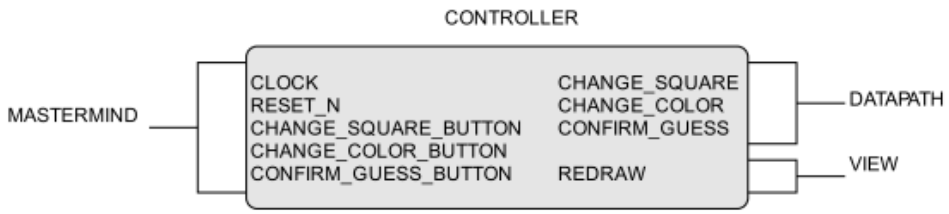
Al controller è affidato il compito della gestione dei segnali e degli eventi possibili esclusivamente durante la partita: si occupa, dati in ingresso i segnali di cambio colore, *posizione* e **conferma**, di inviare al datapath e alla view i rispettivi cambiamenti da apportare sia a livello di modello sia a livello grafico. Il controller è gestito unicamente con segnali d'input/output, senza l'uso di stati ma con coppie di variabili che indicano il valore attuale e quello precedente dei dati da gestire e inviare in uscita.

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use work.base_package.all;


entity controller is
 port
 (
  CLOCK                : in  std_logic;
  RESET_N              : in  std_logic;

  CHANGE_SQUARE_BUTTON : in  std_logic;
  CHANGE_COLOR_BUTTON  : in  std_logic;
  CONFIRM_GUESS_BUTTON : in  std_logic;

  -- Connections with Datapath
  CHANGE_SQUARE        : out std_logic;
  CHANGE_COLOR         : out std_logic;
  CONFIRM_GUESS        : out std_logic;

  -- Connections with View
  REDRAW               : out std_logic
 );

end entity;


architecture RTL of controller is
begin

 process (CLOCK, RESET_N)
  variable first_time        : std_logic;
  variable confirm_guess_old : std_logic;
  variable change_square_old : std_logic;
  variable change_color_old  : std_logic;
 begin
  if (RESET_N = '0') then
   CHANGE_SQUARE     <= '0';
   CHANGE_COLOR      <= '0';
   CONFIRM_GUESS     <= '0';
   REDRAW            <= '0';
   first_time        := '1';
   confirm_guess_old := '0';
   change_square_old := '0';
   change_color_old  := '0';
```

Al reset tutto è settato a 0 tranne la variabile first_time che viene di conseguenza settata ad 1 in attesa dell'inizio di una nuova partita.

```
  elsif rising_edge(CLOCK) then
   CHANGE_SQUARE   <= '0';
   CHANGE_COLOR    <= '0';
   CONFIRM_GUESS   <= '0';
   REDRAW          <= '0';
   if (first_time = '1') then
    first_time := '0';
    REDRAW <= '1';
   elsif (change_square_old = '0' and CHANGE_SQUARE_BUTTON =
'1') then
    CHANGE_SQUARE <= '1';
    REDRAW        <= '1';
   elsif (change_color_old = '0' and CHANGE_COLOR_BUTTON =
'1') then
    CHANGE_COLOR  <= '1';
    REDRAW        <= '1';
   elsif (confirm_guess_old = '0' and CONFIRM_GUESS_BUTTON =
'1') then
    CONFIRM_GUESS <= '1';
    REDRAW        <= '1';
   end if;
   confirm_guess_old := CONFIRM_GUESS_BUTTON;
   change_square_old := CHANGE_SQUARE_BUTTON;
   change_color_old  := CHANGE_COLOR_BUTTON;
  end if;
 end process;

end architecture;
```

Una volta avviato il gioco (first_time=1) s'invia il primo segnale di REDRAW alla view e di conseguenza first_time=0; in seguito in base alle *condition*, composte da input e variabili interne, si inviano i diversi segnali in output al datapath e alla view.



X Interfaccia Controller

3.6 Model: datapath e stati della macchina

Il Model è la componente dedicata all'acceso ai dati, la macchina a stati finiti che consente di progettare il gioco, inizializzandolo e mantenendone lo stato in memoria; inoltre crea il necessario livello di astrazione tra il formato in cui i dati sono memorizzati alla fonte e il formato in cui la view si aspetta di riceverli.

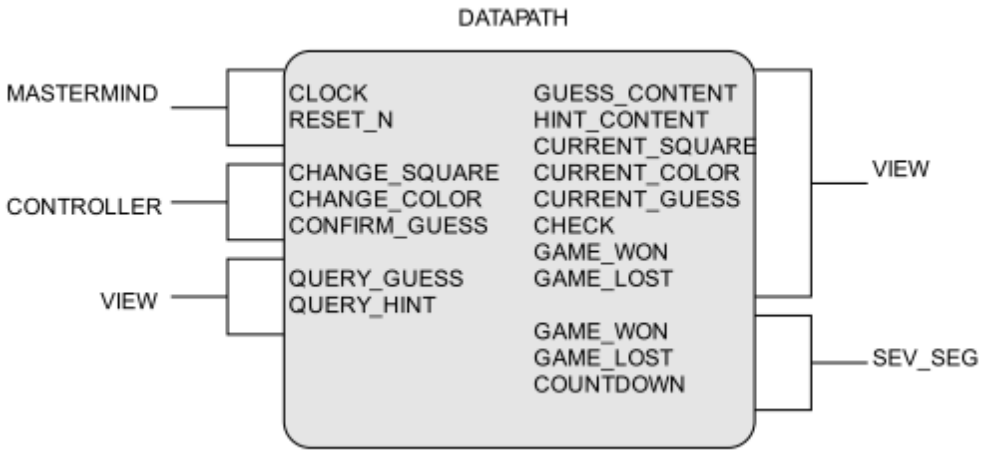
Oltre alla gestione dei dati, il datapath notifica alla view i cambiamenti necessari e, nel nostro caso, riceve da essa anche informazioni riguardanti i tentativi di gioco correnti e i corrispondenti suggerimenti:

- QUERY_GUESS, inviato dalla view, indica il numero del tentativo che la view sta effettivamente rappresentando a video;
- QUERY_HINT, sempre in arrivo dalla view, è il corrispondente valore dei suggerimenti associati al tentativo corrente.

Il datapath definisce diversi campi:

- Inizializza il generatore di valori numerici casuali necessario a generare la sequenza da indovinare
- Indica i cinque stati della macchina
  - *Title*, schermata iniziale e avvio del gioco
  - *Initial*, generazione della sequenza segreta
  - *Play*, partita in corso
  - *SubAns*, elaborazione dei dati
  - *Finished*, fine con vittoria o sconfitta del giocatore
- Definisce le variabili per descrivere il tentativo corrente, il quadrato associato e il colore corrispondente

e di seguito tutti i *signal* necessari.



XI Interfaccia Datapath

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use work.base_package.all;


entity datapath is
 port
 (
  CLOCK          : in  std_logic;
  RESET_N        : in  std_logic;

  -- Connections for the Controller
  CHANGE_SQUARE  : in  std_logic;
  CHANGE_COLOR   : in  std_logic;
  CONFIRM_GUESS  : in  std_logic;

  -- Connections for the View
  QUERY_GUESS     : in  natural range 0 to (GUESSES_NUMBER -
1);
  GUESS_CONTENT : out guess;
  QUERY_HINT   : in  natural range 0 to (GUESSES_NUMBER - 1);
  HINT_CONTENT  : out hint;
  CURRENT_GUESS   : out natural range 0 to (GUESSES_NUMBER -
1);
  CURRENT_SQUARE : out natural range 0 to (SQUARES_NUMBER -
1);
  CURRENT_COLOR  : out guess_peg;
  CHECK          : out guess;
  GAME_WON       : out std_logic;
  GAME_LOST      : out std_logic;
  COUNTDOWN      : out natural

 );
end entity;


architecture RTL of datapath is

 component randomgen
  port(clk:in std_logic;
    a:out std_logic_vector(11 downto 0));
   end component;

 Type StateM is (Title, Initial, Play, SubAns, Finished);
 signal State : StateM;

 signal RandNum : std_logic_vector(11 downto 0);

 shared variable current_guess_r  : integer;
 shared variable current_square_r    : integer;
 shared variable current_color_r     : guess_peg;

 signal check_r             : guess;

 signal guess_board   : guess_board;

 signal hint_board    : hint_board;

 signal countdown_r : natural range 0 to GUESSES_NUMBER;
```

```
function rand_vector_to_guess_peg(x : std_logic_vector)
 return guess_peg is
begin
 return to_integer(unsigned(x));
end function;
```

La macchina modificherà il suo stato in base ai segnali in arrivo, al reset si entra nello stato Title e tutti i valori (counter, colori, suggerimenti,…) sono settati a 0; dal clock successivo, della scheda, per uno qualsiasi dei segnali ricevuti dal controller il sistema va temporaneamente in Initial: qui si genera la combinazione random di colori, nient'altro che bit generati dal random generator, e si va automaticamente nello stato Play; è uno stato fantasma, non visibile a livello concettuale nel diagramma di flusso, che riflette la minima attesa per la generazione della sequenza da parte di un secondo giocatore immaginario.

Di seguito riportiamo e descriviamo il codice degli stati Play e SubAns.

```
                                    when Play =>
       if(CHANGE_SQUARE = '1') then
        if current_square_r + 1 = SQUARES_NUMBER then
         current_square_r := 0;
        else
         current_square_r := (current_square_r + 1);
        end if;
        current_color_r                                          :=
guess_board(current_guess_r)(current_square_r);
       end if;

       if(CHANGE_COLOR = '1') then
        if (current_color_r + 1 = COLORS_NUMBER) then
         current_color_r := 0;
        else
         current_color_r := current_color_r + 1;
        end if;
        guess_board(current_guess_r)(current_square_r)      <=
current_color_r;
       end if;

       if (CONFIRM_GUESS = '1') then
        countdown_r <= countdown_r - 1;
        current_square_r := 0;
        current_color_r := 0;
        State <= SubAns;
       end if;

      when SubAns =>
       if (guess_board(current_guess_r)(0) = check_r(0)) then
        r_spot_count := r_spot_count + 1;
       elsif (guess_board(current_guess_r)(1) = check_r(0) or
         guess_board(current_guess_r)(2) = check_r(0) or
         guess_board(current_guess_r)(3) = check_r(0)) then
        r_color_count := r_color_count + 1;
       end if;
```

```
    if (guess_board(current_guess_r)(1) = check_r(1)) then
     r_spot_count := r_spot_count + 1;
    elsif (guess_board(current_guess_r)(0) = check_r(1) or
      guess_board(current_guess_r)(2) = check_r(1) or
      guess_board(current_guess_r)(3) = check_r(1)) then
     r_color_count := r_color_count + 1;
    end if;

    if (guess_board(current_guess_r)(2) = check_r(2)) then
     r_spot_count := r_spot_count + 1;
    elsif (guess_board(current_guess_r)(0) = check_r(2) or
      guess_board(current_guess_r)(1) = check_r(2) or
      guess_board(current_guess_r)(3) = check_r(2)) then
     r_color_count := r_color_count + 1;
    end if;

    if (guess_board(current_guess_r)(3) = check_r(3)) then
     r_spot_count := r_spot_count + 1;
    elsif (guess_board(current_guess_r)(0) = check_r(3) or
      guess_board(current_guess_r)(1) = check_r(3) or
      guess_board(current_guess_r)(2) = check_r(3)) then
     r_color_count := r_color_count + 1;
    end if;

    for i in 0 to SQUARES_NUMBER - 1 loop
     if i < r_spot_count then
      hint_board(current_guess_r)(i) <= PLACE;
     elsif i < r_spot_count + r_color_count then
      hint_board(current_guess_r)(i) <= COLOR;
     else
      hint_board(current_guess_r)(i) <= NOTHING;
     end if;
    end loop;

    current_guess_r := current_guess_r + 1;

    if (r_spot_count = 4) then
     GAME_WON <= '1';
     State <= Finished;
    elsif (countdown_r = 0) then
     GAME_LOST <= '1';
     State <= Finished;
    else
     r_spot_count  := 0;
     r_color_count := 0;
     State <= Play;
    end if;

  when Finished =>

  when others =>
   State <= Initial;

end case;
```

Nello stato Play, in base ai segnali ricevuti dal controller, si modificano rispettivamente il quadrato corrente, il colore corrente e si accetta la conferma della combinazione: esclusivamente raggiungendo l'ultimo caso, premendo quindi il tasto fisico corrispondente a "*conferma combinazione*", si entra nello stato SubAns per l'elaborazione dei dati.

Il datapath utilizza lo stesso algoritmo condizionale per controllare se la sequenza inserita e confermata dall'utente corrisponda a quella segreta, in particolare per ogni quadrato:

- ✓ prima controlla se alla posizione specifica corrisponde l'esatto colore indicato dall'utente e in caso positivo il contatore *r_spot_count* viene incrementato di uno;
- ✓ in caso contrario si controlla se lo specifico colore è presente negli altri, ciò serve per inviare il feedback "colore giusto, posto sbagliato", in questa ipotesi s'incrementa il contatore *r_color_count* di uno.

Una volta effettuata la verifica, nel caso il giocatore debba proseguire la partita, è necessario inviare alla view i feedback da mostrare all'utente: per ogni quadrato 0,1,2,3 se il posto è esatto con colore esatto inviamo PLACE, nel caso di solo colore esatto in posto errato viene inviato COLOR altrimenti il quadrato rimarrà vuoto. Tutto è gestito con contatori.

A questo punto se il valore del contatore r_spot_count è uguale a quattro, quindi la sequenza è esatta, s'invia il feedback alla view e si va nello stato Finished, allo stesso modo nel caso in cui i tentativi siano terminati e la partita risulti di conseguenza persa; nel caso non ci si trovi in nessuna di queste condizioni i contatori si azzerano e si continua a giocare con una nuova sequenza da inserire.

Lo stato Finished è vuoto, in questo stato vi si resta fino a quando non è eseguito il reset tramite switch: nulla è controllato, settato o elaborato ma risulta necessario in quanto, se si rimanesse sullo stato attuale, ad ogni clock verrebbe controllata una sequenza già controllata, cosa che risulta dispendiosa e inutile.

In qualsiasi altro caso di errore o imprevisto si torna allo stato Initial, si genera una nuova sequenza e si avvia la partita.

3.7 View

La view è l'ultimo componente da analizzare nell'ambito del pattern MVC utilizzato in questo progetto; in input riceve il segnale REDRAW dal controller e tutti i segnali out precedentemente osservati nel model, inoltre la comunicazione si estende e sono introdotti segnali con la VGA: in input si riceve un segnale di ready mentre in output si inviano le indicazioni sul testo, sulle sequenze da disegnare e le coordinate delle posizioni associate ad ogni colore, tutto da e verso VGA_framebuffer.

L'architettura è costituita da:

- un insieme di costanti per definire i margini, la risoluzione e i colori di background prefissati nell'applicazione;
- vengono poi associati i singoli colori disponibili nel gioco a numeri in ordine crescente da 0 a 7;
- infine si definiscono i colori associati al feedback, ossia ai suggerimenti osservabili a video dall'utente.

Le funzioni guess_peg_to_color e hint_peg_to_color permettono la corretta gestione dei colori, sia per la sequenza sia per i suggerimenti, in modo da realizzare il concetto di enumerativo e poter associare a ogni colore ricevuto dal datapath un colore appartenente rispettivamente a GUESS_PEG_COLORS e HINT_PEG_COLORS instanziate nella view.

Similmente al datapath, la view è composta di stati e sottostati.

1) IDLE: in caso di reset si è in questo stato, tutti i segnali per la VGA sono settati a 0 tranne initial_screen che va automaticamente ad 1 e permette di avviare il video; il substate resta in CLEAR_SCENE per tutto il tempo in cui lo stato principale è IDLE.
2) WAIT_FOR_READY: si trasferisce in questo stato dallo stato idle quando viene ricevuto il segnale REDRAW da parte del controller e vi si resta fino a quando non si riceve il segnale READY da parte della VGA, specificatamente dal VGA_framebuffer.
3) DRAWING: in questo stato distinguiamo quattordici substate, uno per ogni diversa operazione ossia per ogni differente componente da disegnare.

Di seguito inseriamo il codice nel caso la macchina si trovi nello stato DRAWING, focalizzandoci sui relativi sottostati e su cosa rappresenti ognuno di essi.

```
    when DRAWING =>
     state <= WAIT_FOR_READY;

     case (substate) is
      when CLEAR_SCENE =>
       FB_COLOR <= BACKGROUND_COLOR;
       FB_CLEAR <= '1';
       if initial_screen = '1' then
        initial_screen := '0';
        substate <= DRAW_INITIAL_TEXT;
       elsif GAME_WON = '1' then
        substate <= DRAW_GAME_WON;
       elsif GAME_LOST = '1' then
        substate <= DRAW_GAME_LOST;
       else
        substate <= DRAW_GUESS_BOARD;
       end if;

      when DRAW_INITIAL_TEXT =>
       FB_COLOR       <= TEXT_COLOR;
       FB_TEXT        <= TITLE;
       FB_X0               <= TEXT_X - (TITLE_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y0          <= TEXT_Y - FONT_HEIGHT / 2;
       FB_X1               <= TEXT_X + (TITLE_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y1          <= TEXT_Y + FONT_HEIGHT / 2;
       FB_DRAW_TEXT   <= '1';
       substate       <= DRAW_INTRO_TEXT;

      when DRAW_INTRO_TEXT =>
       FB_COLOR       <= TEXT_COLOR;
       FB_TEXT        <= INTRO;
       FB_X0               <= TEXT_X - (INTRO_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y0                <= TEXT_Y - FONT_HEIGHT / 2 +
FONT_HEIGHT * 4;
       FB_X1               <= TEXT_X + (INTRO_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y1                <= TEXT_Y + FONT_HEIGHT / 2 +
FONT_HEIGHT * 4;
       FB_DRAW_TEXT   <= '1';
       substate       <= FLIP_FRAMEBUFFER;

      when DRAW_GAME_WON =>
       FB_COLOR       <= TEXT_COLOR;
       FB_TEXT        <= WIN;
       FB_X0                <= TEXT_X - (WIN_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y0          <= TEXT_Y - FONT_HEIGHT / 2;
       FB_X1                <= TEXT_X + (WIN_TEXT'length *
FONT_WIDTH) / 2;
       FB_Y1          <= TEXT_Y + FONT_HEIGHT / 2;
       FB_DRAW_TEXT   <= '1';
       substate       <= DRAW_CHECK_GUESS;

      when DRAW_GAME_LOST =>
       FB_COLOR       <= TEXT_COLOR;
       FB_TEXT        <= LOSE;
       FB_X0               <= TEXT_X - (LOSE_TEXT'length *
FONT_WIDTH) / 2;
```

```
        FB_Y0          <= TEXT_Y - FONT_HEIGHT / 2;
        FB_X1               <= TEXT_X + (LOSE_TEXT'length *
FONT_WIDTH) / 2;
        FB_Y1          <= TEXT_Y + FONT_HEIGHT / 2;
        FB_DRAW_TEXT   <= '1';
        substate       <= DRAW_CHECK_GUESS;

      when DRAW_CHECK_GUESS =>
        FB_COLOR                                              <=
guess_peg_to_color(CHECK(square_to_draw));
        FB_X0        <= TEXT_X - (SQUARES_NUMBER * SQUARE_SIZE
+ (SQUARES_NUMBER) * SQUARE_SPACING) / 2
            +  SQUARE_SPACING  *  (square_to_draw  +  1)  +
SQUARE_SIZE * square_to_draw;
        FB_Y0        <= TEXT_Y + FONT_HEIGHT * 2;
        FB_X1        <= TEXT_X - (SQUARES_NUMBER * SQUARE_SIZE
+ (SQUARES_NUMBER - 1) * SQUARE_SPACING) / 2
            +  SQUARE_SPACING  *  (square_to_draw  +  1)  +
SQUARE_SIZE * (square_to_draw + 1);
        FB_Y1               <=  TEXT_Y  +  FONT_HEIGHT  *  2  +
SQUARE_SPACING + SQUARE_SIZE;
        FB_DRAW_FILLED_RECT <= '1';
        if (square_to_draw + 1 < SQUARES_NUMBER) then
         square_to_draw <= square_to_draw + 1;
         substate       <= DRAW_CHECK_GUESS;
        else
         square_to_draw <= 0;
         substate <= FLIP_FRAMEBUFFER;
        end if;

      when DRAW_GUESS_BOARD =>
        FB_COLOR     <= TABLES_COLOR;
        FB_X0        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER;
        FB_Y0        <= TOP_MARGIN + TOP_SPACING_TO_CENTER;
        FB_X1        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER
            + SQUARES_NUMBER * (SQUARE_SIZE + SQUARE_SPACING)
            + SQUARE_SPACING;
        FB_Y1         <= TOP_MARGIN  + TOP_SPACING_TO_CENTER +
GUESSES_NUMBER   *   (SQUARE_SIZE   +   SQUARE_SPACING)   +
SQUARE_SPACING;
        FB_DRAW_FILLED_RECT <= '1';
        substate     <= DRAW_GUESSES;

      when DRAW_GUESSES =>
        FB_COLOR                                              <=
guess_peg_to_color(GUESS_CONTENT(square_to_draw));
        FB_X0        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER +
SQUARE_SPACING  *  (square_to_draw  +  1)  +  SQUARE_SIZE  *
square_to_draw;
        FB_Y0          <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING  *  (query_guess_r  +  1)  +  SQUARE_SIZE  *
query_guess_r;
        FB_X1        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER +
SQUARE_SPACING  *  (square_to_draw  +  1)  +  SQUARE_SIZE  *
(square_to_draw + 1);
        FB_Y1          <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING  *  (query_guess_r  +  1)  +  SQUARE_SIZE  *
(query_guess_r + 1);
        FB_DRAW_FILLED_RECT <= '1';
        if (square_to_draw + 1 < SQUARES_NUMBER) then
         square_to_draw <= square_to_draw + 1;
```

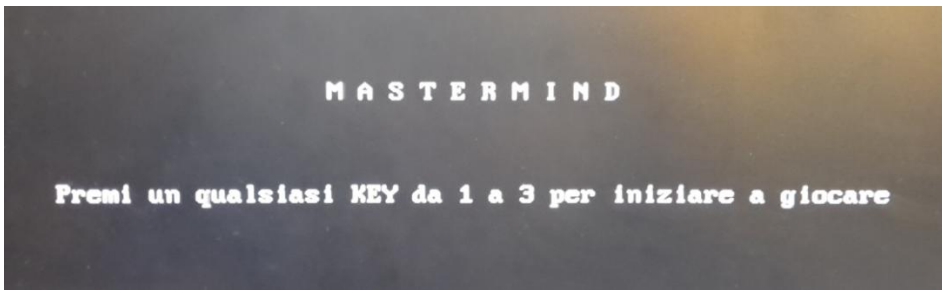
```
         substate       <= DRAW_GUESSES;
        else
         square_to_draw <= 0;
         if (query_guess_r + 1 <= CURRENT_GUESS) then
          query_guess_r <= query_guess_r + 1;
          substate    <= DRAW_GUESSES;
         else
          query_guess_r <= 0;
          substate    <= DRAW_HINT_BOARD;
         end if;
        end if;

       when DRAW_HINT_BOARD =>
        FB_COLOR     <= TABLES_COLOR;
        FB_X0                      <=   LEFT_MARGIN   *   2   +
LEFT_SPACING_TO_CENTER  +  SQUARES_NUMBER  *  (SQUARE_SIZE  +
SQUARE_SPACING) + SQUARE_SPACING;
        FB_Y0        <= TOP_MARGIN + TOP_SPACING_TO_CENTER;
        FB_X1                      <=   LEFT_MARGIN   *   2   +
LEFT_SPACING_TO_CENTER  +  (SQUARES_NUMBER  *  (SQUARE_SIZE  +
SQUARE_SPACING)) * 2 + SQUARE_SPACING * 2;
        FB_Y1         <= TOP_MARGIN  + TOP_SPACING_TO_CENTER +
GUESSES_NUMBER    *    (SQUARE_SIZE    +    SQUARE_SPACING)    +
SQUARE_SPACING;
        FB_DRAW_FILLED_RECT <= '1';
        if CURRENT_GUESS > 0 then
         substate <= DRAW_HINTS;
        else
         substate <= DRAW_SQUARE_BORDER;
        end if;

       when DRAW_HINTS =>
        FB_COLOR                                                <=
hint_peg_to_color(HINT_CONTENT(square_to_draw));
        FB_X0                      <=   LEFT_MARGIN   *   2   +
LEFT_SPACING_TO_CENTER  +  SQUARES_NUMBER  *  (SQUARE_SIZE  +
SQUARE_SPACING) + SQUARE_SPACING
           +   SQUARE_SPACING   *   (square_to_draw   +   1)   +
SQUARE_SIZE * square_to_draw;
        FB_Y0          <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING   *   (query_hint_r   +   1)   +   SQUARE_SIZE   *
query_hint_r;
        FB_X1                      <=   LEFT_MARGIN   *   2   +
LEFT_SPACING_TO_CENTER  +  SQUARES_NUMBER  *  (SQUARE_SIZE  +
SQUARE_SPACING)
           +   SQUARE_SPACING   *   (square_to_draw   +   2)   +
SQUARE_SIZE * (square_to_draw + 1);
        FB_Y1          <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING   *   (query_hint_r   +   1)   +   SQUARE_SIZE   *
(query_hint_r + 1);
        FB_DRAW_FILLED_RECT <= '1';
        if   (square_to_draw   +   1   <   SQUARES_NUMBER)   and
(HINT_CONTENT(square_to_draw + 1) /= NOTHING) then
         square_to_draw <= square_to_draw + 1;
         substate       <= DRAW_HINTS;
        else
         square_to_draw <= 0;
         if (query_hint_r + 1 < CURRENT_GUESS) then
          query_hint_r  <= query_hint_r + 1;
          substate    <= DRAW_HINTS;
         else
```

```
          query_hint_r <= 0;
          substate    <= DRAW_SQUARE_BORDER;
         end if;
        end if;

       when DRAW_SQUARE_BORDER =>
        FB_COLOR <= SELECTION_COLOR;
        FB_X0        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER
              + SQUARE_SPACING * CURRENT_SQUARE
              + SQUARE_SIZE * CURRENT_SQUARE;
        FB_Y0         <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING * CURRENT_GUESS
              + SQUARE_SIZE * CURRENT_GUESS;
        FB_X1        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER +
              SQUARE_SPACING * (CURRENT_SQUARE + 2)
              + SQUARE_SIZE * (CURRENT_SQUARE + 1);
        FB_Y1         <= TOP_MARGIN + TOP_SPACING_TO_CENTER +
SQUARE_SPACING * (CURRENT_GUESS + 2)
              + SQUARE_SIZE * (CURRENT_GUESS + 1);
        FB_DRAW_EMPTY_RECT <= '1';
        substate <= DRAW_SELECTABLE_COLORS;

       when DRAW_SELECTABLE_COLORS =>
        FB_COLOR     <= GUESS_PEG_COLORS(square_to_draw);
        FB_X0        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER +
(SQUARE_SPACING + LEFT_MARGIN) / 2
              + SQUARE_SPACING * (square_to_draw + 1)
              + SQUARE_SIZE * square_to_draw;
        FB_Y0         <= TOP_MARGIN  + TOP_SPACING_TO_CENTER *
3
             + GUESSES_NUMBER * (SQUARE_SIZE + SQUARE_SPACING)
             + SQUARE_SPACING;
        FB_X1        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER
              + (SQUARE_SPACING + LEFT_MARGIN) / 2
              + SQUARE_SPACING * (square_to_draw + 1)
              + SQUARE_SIZE * (square_to_draw + 1);
        FB_Y1         <= TOP_MARGIN  + TOP_SPACING_TO_CENTER *
3
             + GUESSES_NUMBER * (SQUARE_SIZE + SQUARE_SPACING)
             + SQUARE_SPACING + SQUARE_SIZE;
        FB_DRAW_FILLED_RECT <= '1';
        if (square_to_draw + 1 < COLORS_NUMBER) then
         square_to_draw <= square_to_draw + 1;
         substate       <= DRAW_SELECTABLE_COLORS;
        else
         square_to_draw <= 0;
         substate    <= DRAW_SELECTED_COLOR_BORDER;
        end if;

       when DRAW_SELECTED_COLOR_BORDER =>
        FB_COLOR <= SELECTION_COLOR;
        FB_X0        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER +
(SQUARE_SPACING + LEFT_MARGIN) / 2
              + SQUARE_SPACING * CURRENT_COLOR
              + SQUARE_SIZE * CURRENT_COLOR;
        FB_Y0         <= TOP_MARGIN  + TOP_SPACING_TO_CENTER *
3
             + GUESSES_NUMBER * (SQUARE_SIZE + SQUARE_SPACING)
;
        FB_X1        <= LEFT_MARGIN + LEFT_SPACING_TO_CENTER
              + (SQUARE_SPACING + LEFT_MARGIN) / 2
```

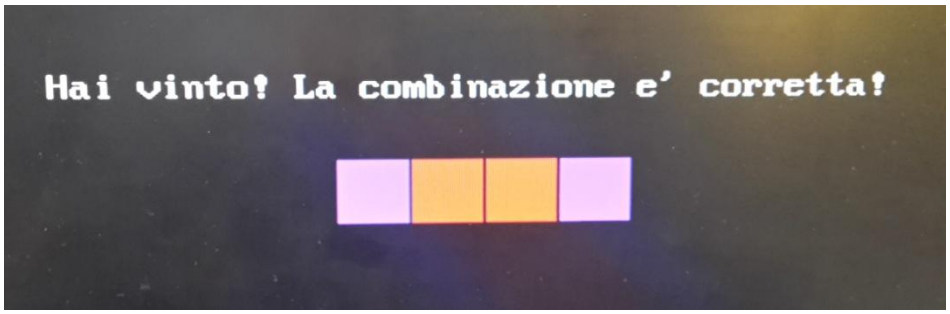
```
                + SQUARE_SPACING * (CURRENT_COLOR + 2)
                + SQUARE_SIZE * (CURRENT_COLOR + 1);
            FB_Y1          <= TOP_MARGIN  + TOP_SPACING_TO_CENTER *
3
                + GUESSES_NUMBER * (SQUARE_SIZE + SQUARE_SPACING)
                + SQUARE_SPACING * 2 + SQUARE_SIZE;
            FB_DRAW_EMPTY_RECT <= '1';
            substate <= FLIP_FRAMEBUFFER;

          when FLIP_FRAMEBUFFER =>
            FB_FLIP  <= '1';
            state    <= IDLE;

        end case;
    end case;
```

Si riportano le principali azioni compiute in ogni sottostato, raggruppando quelli con funzionalità simili e ricordando che lo scambio di segnali è fra view, datapath e VGA_framebuffer.

- CLEAR_SCENE, utilizzato molteplici volte, con background nero, a partire dai segnali può traslare in 4 diversi substate
    - DRAW_INITIAL_TEXT con schermata iniziale del gioco **MASTERMIND,** che a sua volta chiama il sottostato
        - DRAW_INTRO_TEXT con indicazioni su come iniziare una nuova partita

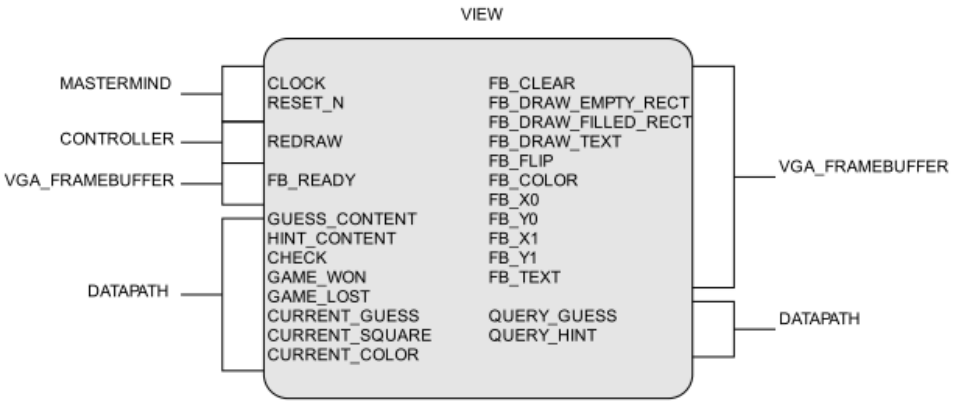


    - DRAW_GAME_WON / DRAW_GAME_LOST forniscono a video il testo con indicazione di vittoria/sconfitta e richiamano il sottostato
        - DRAW_CHECK_GUESS che mostra la sequenza segreta



    - DRAW_GUESS_BOARD che disegna la prima metà della board di gioco in cui inserire la sequenza di gioco desiderata
- DRAW_HINT_BOARD per tracciare l'altra metà di board in cui appariranno i suggerimenti;
- DRAW_GUESSES per disegnare i tentativi finora inseriti dall'utente, compreso quello attuale non ancora confermato;

- DRAW_HINTS per disegnare i suggerimenti generati fino a questo momento;
- DRAW_SQUARE_BORDER per disegnare il bordo del quadrato attualmente selezionato;
- DRAW_SELECTABLE_COLORS disegna la riga contenente l'elenco di colori selezionabili, rappresentati anch'essi sotto forma di quadrati, e si trasferisce nel substate
    - DRAW_SELECTED_COLOR_BORDER che evidenzia il colore del quadrato che l'utente sta attualmente modificando;
- FLIP_FRAMEBUFFER si utilizza per la gestione della lettura e scrittura in SRAM, in particolare permette di traslare, nella memoria fisica, dalla parte alta a quella bassa e viceversa

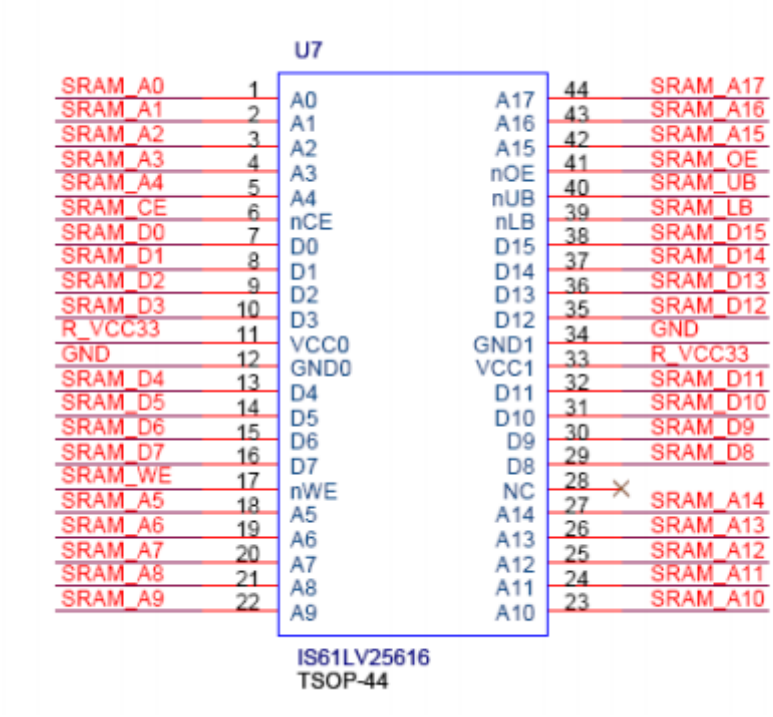


XII Interfaccia View

3.8 Interfacciamento alla memoria

La memoria SRAM, acronimo di Static Random Access Memory, è una memoria RAM volatile, include array rettangolari di celle di memoria e un circuito di supporto per la decodifica degli indirizzi e per l'implementazione delle operazioni di lettura/scrittura. L'array è organizzato in righe e colonne di celle di memoria (WORD LINE e BIT LINE): ciascuna cella ha un indirizzo univoco definito dall'intersezione riga/colonna.

La SRAM presente nella scheda DE1 necessita di 39 linee I/O per dialogare con l'unità di controllo: 18 linee di input per l'assegnazione dell'indirizzo (A0-A17), 16 piedini di I/O per la lettura/scrittura dei dati (I/O0-I/O15) più 5 linee di input per il controllo dei tipici segnali presenti nelle memorie più note: Chip Enable, Output Enable, Write Enable, Lower Byte, Upper Byte. Oltre a questi sono presenti naturalmente i piedini di alimentazione e di massa.



XIII SRAM Schema

La SRAM è una memoria a porta singola e ciò significa che può svolgere una sola operazione alla volta, o lettura o scrittura, quindi abbiamo bisogno di disaccoppiare i canali (Scheda/VGA) e renderli indipendenti: la memoria fisica è una sola, pertanto sarà necessario predisporre un arbitro che impedisca situazioni di starvation.

La partizione virtuale consiste nel mantenere in memoria la "scena", in particolare: in un'area della SRAM vado a scrivere (disegnare) la scena successiva, quella da visualizzare al successivo flip del framebuffer, mentre nell'altra si legge la scena da visualizzare a video; questo processo è ciclico e le parti si invertono in presenza del segnale di FLIP.

Nel dettaglio le entità che scambiano segnali con la SRAM sono:

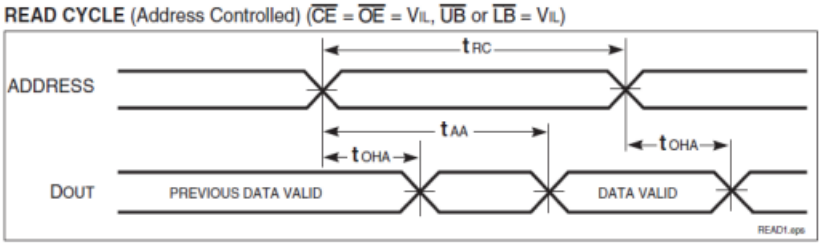
- VGA_framebuffer,
- Mastermind,
- VGA_ramdac;

le prime due si occupano esclusivamente dei segnali da e verso la memoria, senza effettivi accessi in essa, avviando una gestione annidata che dall'entità mastermind invoca framebuffer che a sua volta richiama la VGA_ramdac.

In VGA_ramdac si legge e scrive effettivamente dalla/nella memoria attraverso l'uso di segnali collegati ai pin fisici:

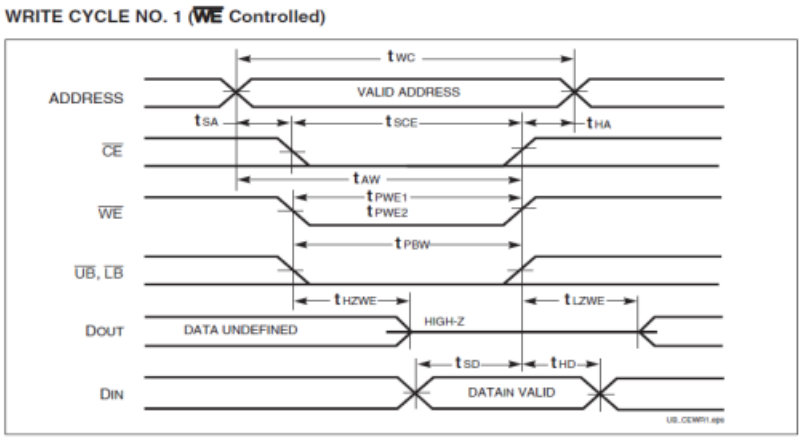
- CE *Chip Enable* per abilitare e disabilitare il dispositivo
- WE *Write Enable* per abilitare la scrittura in memoria
- OE *Output Enable* abilita l'uscita per la lettura
- LB e UB *Lower* e *Upper byte* control
- ADDR per gli input *Address*

Per comandare la memoria in modo che possa essere letta, dovremmo abilitare il CE, OE, LB (o UB) mentre dovremmo tenere disabilitato il WE; essendo segnali attivi bassi, per abilitarli devono essere posti uguali a zero.



XIV Forme d'onda relative al ciclo di lettura della memoria

Durante il ciclo di scrittura invece bisogna abilitare il CE, LB (o UB) e disabilitare l'OE; la scrittura sarà comandata con il WE e solo una volta disabilitato sarà possibile cambiare indirizzo.



XV Forme d'onda relative al ciclo di scrittura della memoria

Di seguito i segnali di input/output presenti in VGA_ramdac.

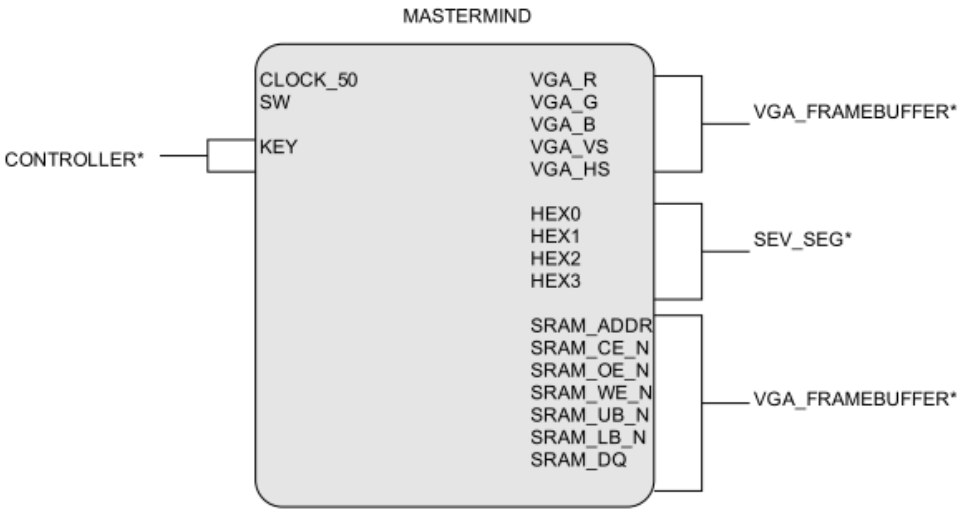
```
SRAM_ADDR           : out   std_logic_vector(17 downto 0);
SRAM_DQ             : inout std_logic_vector(15 downto 0);
SRAM_CE_N           : out   std_logic;
SRAM_OE_N           : out   std_logic;
SRAM_WE_N           : out   std_logic;
SRAM_UB_N           : out   std_logic;
SRAM_LB_N           : out   std_logic
```

3.9 MASTERMIND

Possiamo definire l'entità mastermind come il main dell'intero progetto, gestisce, invoca e invia tutti i segnali necessari a gestire:

- le entità del pattern MVC,
- la vga,
- i display a 7 segmenti.

Di seguito, attraverso le interfacce, è possibile visualizzare tutti i segnali in ingresso e in uscita rispetto a ogni singola entità in gioco.



XVI Interfaccia Mastermind

All'interno del codice associato all'entità mastermind osserviamo

```
pll : entity work.PLL
      port map (
            inclk0  => CLOCK_50,
            c0      => clock_vga,
            c1      => clock
      );
```

PLL (Phase-locked loop) è un circuito elettronico in grado di costituire un sistema di controllo automatico, per generare segnali periodici la cui fase è in relazione fissa con quella del segnale di riferimento; pur avendo diverse finalità, nei sistemi a microprocessore si utilizza principalmente per generare clock ed ottenere una *sincronizzazione costante nel tempo* assorbendo eventuali variazioni nella frequenza del segnale a cui si fa riferimento: si controlla la frequenza in uscita generata e la si modifica, alterando la tensione, fino a quando non coinciderà con la frequenza del segnale in ingresso.

Oltre al sopraelencato, Mastermind si occupa della gestione del reset sincrono, attuabile attraverso lo switch 9 della board e necessario per avviare il gioco.

```
reset_sync : process(CLOCK_50)
begin
      if (rising_edge(CLOCK_50)) then
            reset_sync_reg <= SW(9);
            RESET_N <= reset_sync_reg;
      end if;
end process;
```

3.10 VGA

Per visualizzare un'immagine, il display deve poter ricevere i segnali RGB caratterizzanti i pixel e inserirli uno per volta lungo le righe, partendo dall'angolo in alto a sinistra; raggiunto il termine della prima riga, la coordinata colonna è portata a zero e quella di riga incrementata di uno ciò è ripetuto fino a quando non si raggiungerà l'angolo opposto a quello di partenza. Una volta concluso tale processo il display si dovrà nuovamente aggiornare e il ciclo inizierà da capo.

Anche nel caso delle entità VGA abbiamo tipi e costanti utili nel progetto, in particolare sono elencati i colori composti in RGB e due sottotipi per identificare:

- un integer che distingua una precisa coordinata,
- uno specifico colore fra le costanti gestite.

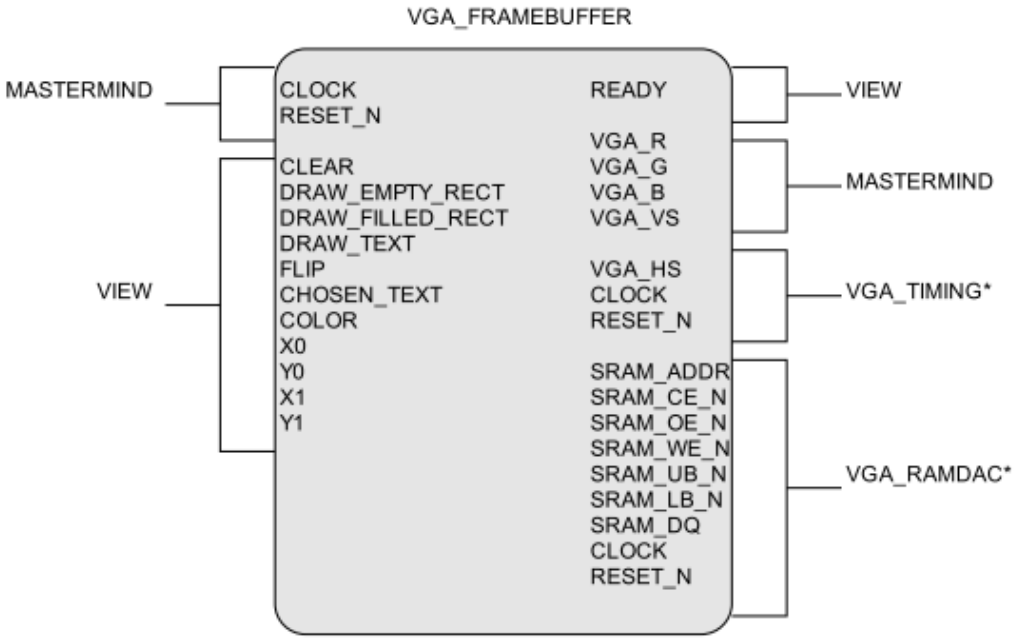
Tutto ciò è definito all'interno del VGA_package di cui riportiamo di seguito il codice.

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

package vga_package is

 subtype   xy_coord_type is integer range 0 to 512;

 subtype   color_type    is std_logic_vector(11 downto 0);

 constant  COLOR_BROWN   : color_type := X"850";

 constant  COLOR_ORANGE  : color_type := X"F80";
 constant  COLOR_RED     : color_type := X"F00";
 constant  COLOR_GREEN   : color_type := X"0F0";
 constant  COLOR_BLUE    : color_type := X"00F";
 constant  COLOR_YELLOW  : color_type := X"FF0";
 constant  COLOR_CYAN    : color_type := X"0FF";
 constant  COLOR_MAGENTA : color_type := X"F0F";
 constant  COLOR_PURPLE  : color_type := X"80F";

 constant  COLOR_BLACK   : color_type := X"000";
 constant  COLOR_WHITE   : color_type := X"FFF";

end package;
```

Di seguito saranno discusse le tre principali entità riferite alla VGA, alla sua gestione, all'interfacciamento con la memoria e alla sincronizzazione.

3.10.1 VGA_FRAMEBUFFER

Questa entità si occupa di gestire l'insieme dei blocchi della VGA, con i dovuti segnali di ingresso/uscita visibili nelle interfacce riportate di seguito.



XVII Interfaccia VGA framebuffer

All'interno dell'architettura possiamo individuare quattro stati:

- IDLE
- DRAWING_FILLED_RECT per disegnare un rettangolo pieno
- DRAWING_EMPTY_RECT per disegnare un rettangolo vuoto
- DRAWING_TEXT per disegnare il carattere del testo

e relativi sottostati in cui traslano per poter inizializzare e disegnare tutto ciò che dovrà essere mostrato a video: rettangoli vuoti, rettangoli pieni colorati e caratteri del testo.

Dallo stato IDLE è possibile traslare nei 3 stati sopraindicati, nonché:

- Pulire la scena (CLEAR)
- Fare lo switch fra le partizioni virtuali della SRAM utilizzata come buffer (FLIP).

Nel secondo caso, il framebuffer salva l'arrivo del segnale di flip e al clock successivo inverte le parti della SRAM da leggere e da scrivere, ciò avviene tramite il segnale fb_buffer_idx, che in VGA_ramdac corrisponderà al segnale BUFFER_INDEX; a questo punto si azzera flip_on_next_vs poiché il framebuffer ha gestito il segnale e potrà iniziare un nuovo ciclo quando verrà riasserito il segnale di REDRAW.

Osserviamo l'esigenza del segnale flip_on_next_vs poiché non sempre il framebuffer può attuare immediatamente lo scambio, ma in molti casi sarà necessario attendere il concludersi della proiezione di una scena, ossia v_sync= '0'.

Di seguito porzione del codice.

```
      fb_rd_x   <= vga_x;
 fb_rd_y   <= vga_y;
 fb_rd_req <= not(vga_blank) and (vga_strobe or fb_rd_ack or
'1');

 fb_wr_color <= latched_color;
 fb_wr_x           <=  std_logic_vector(to_unsigned(x_cursor,
fb_wr_x'length));
 fb_wr_y           <=  std_logic_vector(to_unsigned(y_cursor,
fb_wr_y'length));

 VGA_VS  <= vga_vsync;
 VGA_R    <= fb_rd_color(11 downto 8) when (vga_blank = '0')
else (others => '0');
 VGA_G    <= fb_rd_color(7 downto 4)  when (vga_blank = '0')
else (others => '0');
 VGA_B    <= fb_rd_color(3 downto 0)  when (vga_blank = '0')
else (others => '0');
 READY       <=  '1'  when  (state  =  IDLE  and  (CLEAR  or
DRAW_FILLED_RECT or DRAW_EMPTY_RECT
 or DRAW_TEXT or FLIP) = '0') else '0';

 draw_logic : process(CLOCK, RESET_N)
 begin

  if (RESET_N = '0') then

   state           <= IDLE;
   next_bit       <= '0';
   next_line       <= '0';
   fb_wr_req       <= '0';
   fb_buffer_idx   <= '0';
   flip_on_next_vs <= '0';

  elsif (rising_edge(CLOCK)) then

   fb_wr_req <= '0';

   case (state) is
    when IDLE =>

     latched_color <= COLOR;

     if (CLEAR = '1') then
      x_start   <= 0;
      y_start   <= 0;
      x_end     <= SCREEN_WIDTH-1;
      y_end     <= SCREEN_HEIGHT-1;
      state     <= DRAWING_FILLED_RECT;
      substate  <= INIT;
```

```
 elsif (DRAW_FILLED_RECT = '1') then
  x_start   <= X0;
  y_start   <= Y0;
  x_end     <= X1;
  y_end     <= Y1;
  state     <= DRAWING_FILLED_RECT;
  substate  <= INIT;

 elsif (DRAW_EMPTY_RECT = '1') then
  x_start   <= X0;
  y_start   <= Y0;
  x_end     <= X1;
  y_end     <= Y1;
  state     <= DRAWING_EMPTY_RECT;
  substate  <= INIT;

 elsif (DRAW_TEXT = '1') then
  x_start   <= X0;
  y_start   <= Y0;
  x_end     <= X1;
  y_end     <= Y1;
  state     <= DRAWING_TEXT;
  substate  <= INIT;

 elsif (FLIP = '1') then
  flip_on_next_vs <= '1';
 end if;

 if (flip_on_next_vs = '1' and vga_vsync = '0') then
  fb_buffer_idx   <= not(fb_buffer_idx);
  flip_on_next_vs <= '0';
 end if;

when DRAWING_FILLED_RECT =>
 fb_wr_req <= '1';

 if (substate = INIT) then
  x_cursor  <=  x_start;
  y_cursor  <=  y_start;
  substate  <= DRAWING;
 else
  if (fb_wr_ack = '1') then
   if (x_cursor = x_end) then
    x_cursor <= x_start;
    if (y_cursor = y_end) then
     fb_wr_req <= '0';
     state <= IDLE;
    else
     y_cursor <= y_cursor + 1;
    end if;
   else
    x_cursor <= x_cursor + 1;
   end if;
  end if;
 end if;

when DRAWING_EMPTY_RECT =>
 fb_wr_req <= '1';

 if (substate = INIT) then
```

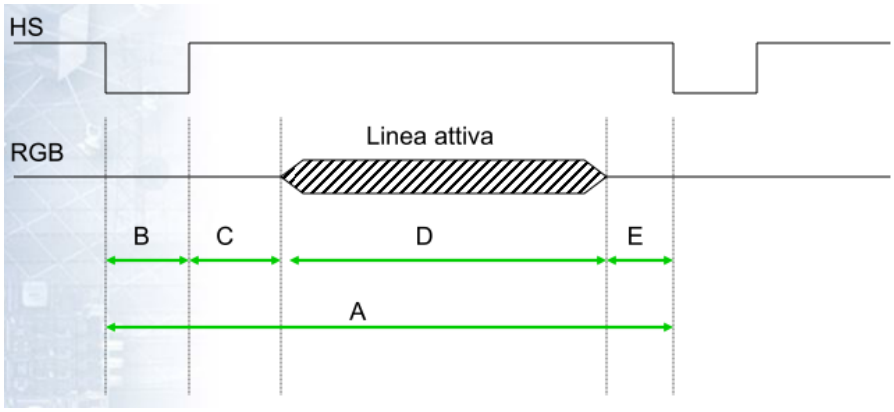
```
 x_cursor  <=  x_start;
 y_cursor  <=  y_start;
 substate  <= DRAWING_HIGH_SIDE;
elsif (substate = DRAWING_HIGH_SIDE) then
 if (fb_wr_ack = '1') then
  if (x_cursor = x_end) then
   x_cursor <= x_start;
   if (y_cursor = y_start + 3) then
    fb_wr_req <= '0';
    y_cursor <= y_cursor + 1;
    substate <= DRAWING_LEFT_SIDE;
   else
    y_cursor <= y_cursor + 1;
   end if;
  else
   x_cursor <= x_cursor + 1;
  end if;
 end if;
elsif (substate = DRAWING_LEFT_SIDE) then
 if (fb_wr_ack = '1') then
  if (x_cursor = x_start + 3) then
   x_cursor <= x_start;
   if (y_cursor = y_end - 4) then
    fb_wr_req <= '0';
    x_cursor <= x_end - 3;
    y_cursor <= y_start + 4;
    substate <= DRAWING_RIGHT_SIDE;
   else
    y_cursor <= y_cursor + 1;
   end if;
  else
   x_cursor <= x_cursor + 1;
  end if;
 end if;
elsif (substate = DRAWING_RIGHT_SIDE) then
 if (fb_wr_ack = '1') then
  if (x_cursor = x_end) then
   x_cursor <= x_end - 3;
   if (y_cursor = y_end - 4) then
    fb_wr_req <= '0';
    x_cursor <= x_start;
    y_cursor <= y_cursor + 1;
    substate <= DRAWING_LOW_SIDE;
   else
    y_cursor <= y_cursor + 1;
   end if;
  else
   x_cursor <= x_cursor + 1;
  end if;
 end if;
else
 if (fb_wr_ack = '1') then
  if (x_cursor = x_end) then
   x_cursor <= x_start;
   if (y_cursor = y_end) then
    fb_wr_req <= '0';
    state <= IDLE;
   else
    y_cursor <= y_cursor + 1;
   end if;
  else
```

```
          x_cursor <= x_cursor + 1;
         end if;
        end if;
       end if;

      when DRAWING_TEXT =>
       if (substate = INIT) then
        x_cursor  <=  x_start;
        y_cursor  <=  y_start;
        substate  <=  DRAWING;
       elsif (substate = WAIT_TEXT_PIXEL) then
        next_bit <= '0';
        next_line <= '0';
        substate <= DRAWING;
       else
        if (text_pixel = '1' and fb_wr_req = '0') then
         fb_wr_req <= '1';
        else
         if (fb_wr_req = '1' and fb_wr_ack = '1') or text_pixel
= '0' then
          fb_wr_req <= '0';
          substate <= WAIT_TEXT_PIXEL;
          if (x_cursor = x_end) then
           x_cursor <= x_start;
           if (y_cursor = y_end) then
            state <= IDLE;
           else
            y_cursor <= y_cursor + 1;
            next_line <= '1';
           end if;
          else
           x_cursor <= x_cursor + 1;
           next_bit <= '1';
          end if;
         end if;
        end if;
       end if;



      when others =>
       assert false severity failure;

      end case;


    end if;

   end process;
```

3.10.2 VGA_TIMING

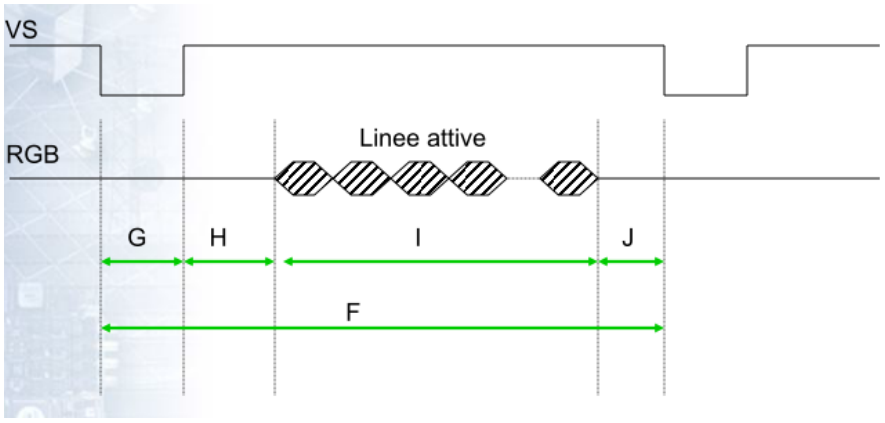
Per visualizzare correttamente un'immagine sul display non basta semplicemente inviare i tre segnali RGB con la giusta frequenza ma sono necessari due segnali di sincronismo:

- H_SYNC: quando l'horizontal sync è asserito (in logica negativa) indica al monitor la fine di una riga.



XVIII Sincronizzazione orizzontale

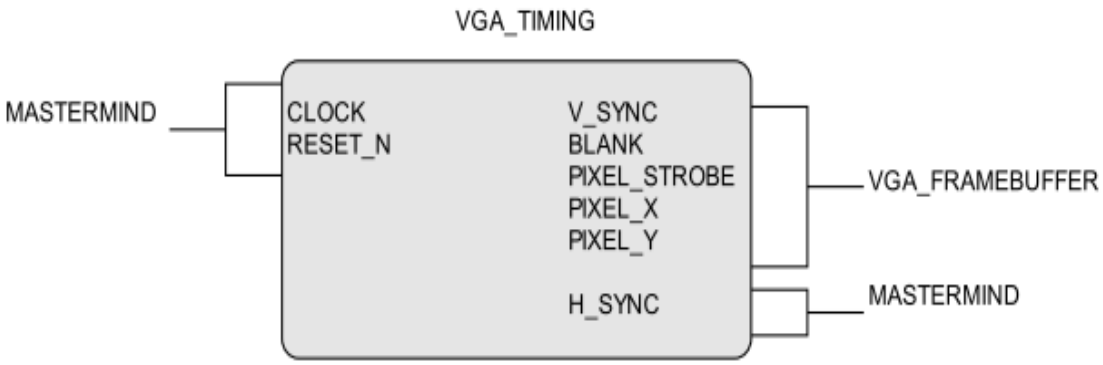
- V_SYNC: quando il vertical sync è asserito (in logica negativa) s'indica al monitor che l'immagine è terminata ed è possibile iniziare un altro ciclo di refresh.



XIX Sincronizzazione verticale

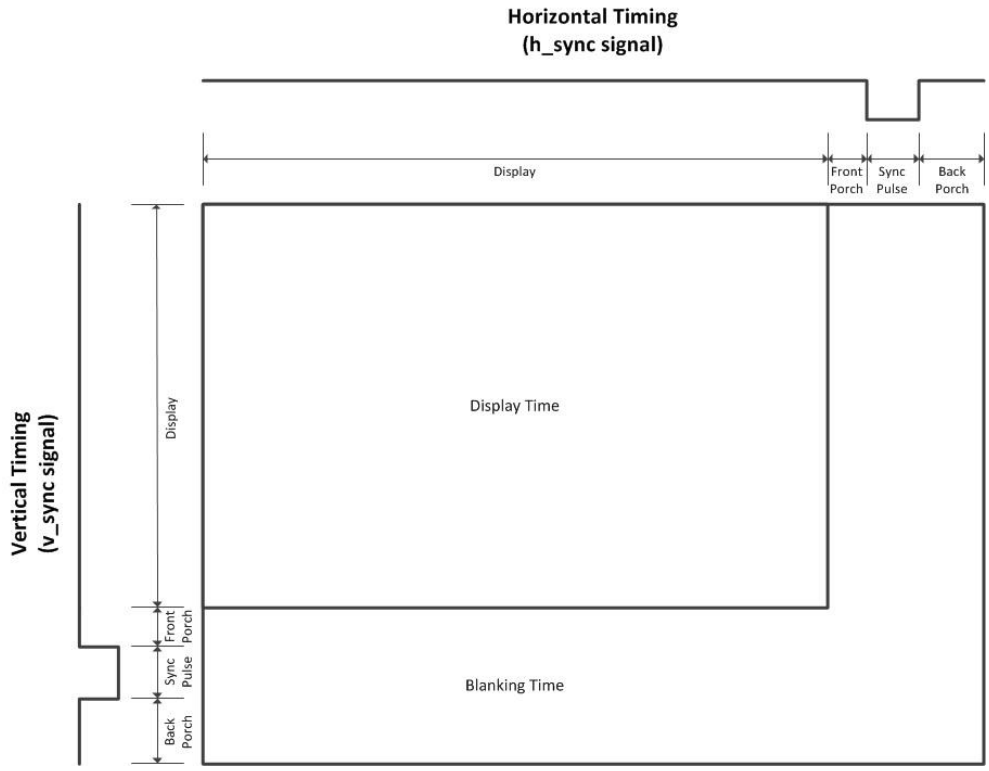
Entrambi i segnali devono rimanere nello stato logico alto mentre i pixel sono disegnati (active region), sono poi portati allo stato logico basso dopo un tempo detto *front porch* (vertical e horizontal) e vi rimangono per un tempo *sync pulse*; infine tornano nello stato iniziale per un tempo *back porch* prima di passare alla nuova riga o alla nuova immagine. La durata dei tempi *sync pulse, front porch e back porch* sia vertical sia horizontal è calcolata a partire dalla risoluzione: l'insieme di tutti questi periodi dà origine alla cosiddetta blanking region.

Nella blanking region i segnali RGB devono essere a "0" e nessun pixel è effettivamente disegnato.



XX Interfaccia VGA Timing

L'entità VGA_timing comunica esclusivamente con VGA inviando in uscita i segnali per la gestione del sincronismo del display e dei singoli pixel.



XXI VGA Timing

Nel nostro progetto l'entità che gestisce la corretta temporizzazione della vga è vga_timing dove, in modo duale, viene gestito sia il sincronismo orizzontale che quello verticale. Nel processo *v_timing* al primo accesso successivo a un RESET_N si resettano sia il *v_counter*, contatore di linea, che il *v_pixel*, contatore di pixel di una linea, e si va a porre il segnale *v_state* nello stato FRONT_PORCH. In seguito ad ogni ciclo di clock, se *new_line* è uguale a '1' si verifica il valore del contatore *v_counter* e, in base ad esso, lo stato del segnale *v_state* traslerà da FRONT_PORCH a DATA passando rispettivamente per gli stati SYNC e BACK_PORCH (vedi figura XXI). Quando *v_state* si trova nello stato DATA e *v_counter* è diverso da '0' si va ad aumentare il contatore "v_pixel" che servirà a settare l'uscita PIXEL_Y utile per disegnare il vero e proprio pixel sullo schermo.

```
v_timing : process(CLOCK, RESET_N)
 begin

  if (RESET_N = '0') then
   v_counter   <= 0;
   v_pixel     <= 0;
   v_state     <= FRONT_PORCH;
  elsif (rising_edge(CLOCK)) then

   if(new_line = '1') then
    if (v_counter = 0) then
     v_state <= FRONT_PORCH;
    elsif (v_counter = (V_FRONT_PORCH - 1)) then
     v_state <= SYNC;
    elsif (v_counter = (V_FRONT_PORCH + V_SYNC_LEN - 1)) then
     v_state <= BACK_PORCH;
    elsif   (v_counter   =   (V_FRONT_PORCH   +   V_SYNC_LEN   +
V_BACK_PORCH - 1)) then
     v_state <= DATA;
    end if;

    if(v_state = DATA and v_counter /= 0) then
     v_pixel <= v_pixel + 1;
    else
     v_pixel <= 0;
    end if;

    if (v_counter = V_LENGTH-1) then
     v_counter <= 0;
    else
     v_counter <= v_counter + 1;
    end if;
   end if;
  end if;

 end process;

 BLANK    <= '0' when (h_state = DATA and v_state = DATA) else
'1';
 H_SYNC   <= '0' when (h_state = SYNC) else '1';
 V_SYNC   <= '0' when (v_state = SYNC) else '1';
 PIXEL_X       <=       std_logic_vector(to_unsigned(h_pixel,
PIXEL_X'LENGTH));
 PIXEL_Y       <=       std_logic_vector(to_unsigned(v_pixel,
PIXEL_Y'LENGTH));
 PIXEL_STROBE <= '1' when (h_state = DATA and v_state = DATA
and clock_count = 0) else '0';


end architecture;
```

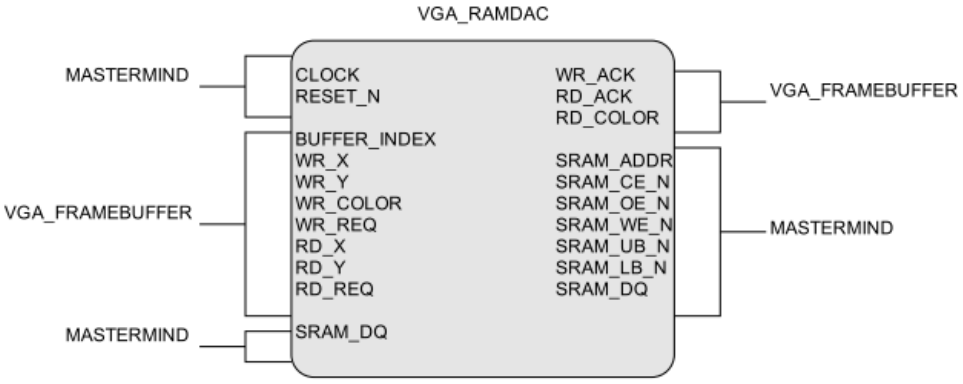
In modo duale viene gestita la visualizzazione dell'immagine relativa all'asse orizzontale di cui sotto mostriamo il codice.

```
h_timing : process(CLOCK, RESET_N)
begin

  if (RESET_N = '0') then
   h_counter   <= 0;
   h_pixel     <= 0;
   h_state     <= FRONT_PORCH;
   new_line    <= '0';
   clock_count <= 0;
  elsif (rising_edge(CLOCK)) then
   new_line    <= '0';

   if (clock_count /= CLOCK_DIV-1) then
    clock_count <= clock_count + 1;
   else
    clock_count <= 0;

    if (h_counter = 0) then
     h_state  <= FRONT_PORCH;
     new_line <= '1';
    elsif (h_counter = (H_FRONT_PORCH - 1)) then
     h_state  <= SYNC;
    elsif (h_counter = (H_FRONT_PORCH + H_SYNC_LEN - 1)) then
     h_state  <= BACK_PORCH;
    elsif   (h_counter   =   (H_FRONT_PORCH   +   H_SYNC_LEN   +
H_BACK_PORCH - 1)) then
     h_state  <= DATA;
    end if;

    if(h_state = DATA and h_counter /= 0) then
     h_pixel <= h_pixel + 1;
    else
     h_pixel <= 0;
    end if;

    if (h_counter = H_LENGTH-1) then
     h_counter <= 0;
    else
     h_counter <= h_counter + 1;
    end if;
   end if;
  end if;

 end process;
```

3.10.3 VGA_RAMDAC

Come anticipato nel capitolo 3.8, il modulo vga_ramdac gestisce gli accessi in scrittura e lettura alla SRAM attraverso il segnale BUFFER_INDEX, proveniente dall'entità vga_framebuffer, che si comporta come il pin di comando di un chip select e in base al valore 0/1 indirizza:

- la parte degli indirizzi della SRAM da leggere e quindi da visualizzare a schermo,
- la parte degli indirizzi dove scrivere la scena, che sarà visualizzata al successivo flip del framebuffer.



XXII Interfaccia VGA Ramdac

Con i segnali *wr_addr* e *rd_addr* si definiscono gli indirizzi attualmente da leggere e da scrivere nella SRAM, inviati da vga_framebuffer e forniti alla funzione *coords_to_addr*: tale funzione è necessaria per ricavare l'effettivo indirizzo della SRAM poiché l'indirizzo in arrivo da vga_framebuffer rappresenta le coordinate della parte della scena attualmente in scrittura o in lettura.

```
function coords_to_addr
 (
  x : std_logic_vector;
  y : std_logic_vector
 )
  return std_logic_vector is
begin
  return y(8 downto 0) & x(8 downto 0);
end function;
```

L'*encoded_pixel* rappresenta il colore da scrivere, è ottenuto dalla funzione *encode_pixel* cui è passato come argomento il segnale WR_COLOR (in ingresso da vga_framebuffer). La funzione definisce tre vettori, che

rappresentano l'RGB, ognuno lungo 4 bit poiché la profondità di colore prescelta è 12 bit; tale modello di colori è diviso per ognuno di questi vettori, in particolare codificheremo ogni colore utilizzando

- 3 bit per il rosso
- 2 bit per il verde
- 3 bit per il blu.

La lettura della scena dalla SRAM è effettuata con la sequenza di passaggi appena descritti ma all'inverso, prima il pixel letto si decifrerà attraverso la funzione decode_pixel (memorizzato nel segnale latched_ram), la quale riassegnerà a ciascuno dei canali di colore 4 bit, ottenendo la profondità di 12 bit, poi inviando il pixel decodificato a RD_COLOR.

```
function   encode_pixel(rgb   :   std_logic_vector(FB_DEPTH-1
downto 0))
  return std_logic_vector
 is
  constant BPC    : natural := FB_DEPTH/3;
  variable red    : std_logic_vector(BPC-1 downto 0);
  variable green  : std_logic_vector(BPC-1 downto 0);
  variable blue   : std_logic_vector(BPC-1 downto 0);
 begin
  blue  := rgb(BPC-1 downto 0);
  green := rgb(BPC*2-1 downto BPC);
  red   := rgb(BPC*3-1 downto BPC*2);

  return red(red'high downto red'high-2)
    & green(green'high downto green'high-1)
    & blue(blue'high downto blue'high-2);
 end function;

function decode_pixel(pixel : std_logic_vector(7 downto 0))
  return std_logic_vector
 is
  constant BPC    : natural := FB_DEPTH/3;
  variable red    : std_logic_vector(BPC-1 downto 0);
  variable green  : std_logic_vector(BPC-1 downto 0);
  variable blue   : std_logic_vector(BPC-1 downto 0);
 begin

  red := (others => pixel(5));
  red(red'high downto red'high-2) := pixel(7 downto 5);

  green := (others => pixel(3));
  green(green'high downto green'high-1) := pixel(4 downto 3);

  blue := (others => pixel(0));
  blue(blue'high downto blue'high-2) := pixel(2 downto 0);

  return red & green & blue;
 end function;
```

In questa entità troviamo tre processi: *ram_fsm, mem_dir_ctrl e ram_regs.*

Il primo gestisce lo stato della ram, in particolare le richieste di lettura e scrittura (RD_REQ e WR_REQ), dove abilitiamo rispettivamente SRAM_OE_N e SRAM_WE_N, con logica negativa, per abilitare la lettura e la scrittura sulla SRAM, con logica positiva RD_ACK e WR_ACK, per segnalare al framebuffer il termine dell'operazione richiesta. Sono inoltre attivati rispettivamente i segnali interni *mem_dir_rd* e *mem_dir_wr*, utili per il secondo processo. Da segnalare che nel caso sia richiesta la lettura, è attivata anche *latch_ram_rd*, anch'essa ci servirà poi.

Il secondo processo gestisce effettivamente la memoria. Nel caso di una lettura, l'operazione avviene attraverso i segnali SRAM_LB_N e SRAM_UB_N che permettono di abilitare l'uscita bassa o l'uscita alta della SRAM, rispettivamente, sul bus SRAM_DQ, da cui sarà possibile leggere il dato. In caso di scrittura l'operazione è simile alla precedente ma nel caso specifico, invece di scrivere solo su una metà del bus, si scrive sulla SRAM_DQ un array da 16 bit composto dalla concatenazione di encoded_pixel; osserviamo che in realtà la scrittura sarà sempre di uno e un solo byte grazie a BUFFER_INDEX che permetterà di selezionare uno solo tra SRAM_LB_N e SRAM_UB_N.

L'ultimo processo si utilizza per gestire lo stato della SRAM, in particolare se il segnale latch_ram_rd, gestito nel primo processo, è attivo allora sul segnale latched_ram si scriverà il contenuto del segnale ram_rd_word, gestito nel secondo processo.

```
begin

 rd_buf_idx     <= BUFFER_INDEX;
 wr_buf_idx     <= not(BUFFER_INDEX);
 wr_addr        <= coords_to_addr(WR_X,WR_Y);
 rd_addr        <= coords_to_addr(RD_X,RD_Y);
 encoded_pixel <= encode_pixel(WR_COLOR);
 RD_COLOR       <= decode_pixel(latched_ram);


 ram_fsm   :   process(RD_REQ,   WR_REQ,   rd_addr,   wr_addr,
encoded_pixel, ram_state, CLOCK)
 begin
  mem_dir_wr       <= '0';
  mem_dir_rd       <= '0';
  RD_ACK           <= '0';
  WR_ACK           <= '0';
  SRAM_OE_N        <= '1';
  SRAM_WE_N        <= '1';
  latch_ram_rd     <= '0';
  next_ram_state   <= ram_state;

  case (ram_state) is

   when IDLE =>

    if (RD_REQ = '1') then
     mem_dir_rd     <= '1';
     SRAM_OE_N      <= '0';
     latch_ram_rd   <= '1';
     RD_ACK         <= '1';

    elsif (WR_REQ = '1') then
     mem_dir_wr     <= '1';
     SRAM_WE_N      <= '0';
     WR_ACK         <= '1';

    end if;

   when others =>
    assert false severity failure;

  end case;

 end process;


 mem_dir_ctrl   :   process(mem_dir_rd,   mem_dir_wr,   rd_addr,
SRAM_DQ, wr_addr, rd_buf_idx, wr_buf_idx, encoded_pixel)
 begin
  SRAM_CE_N        <= '0';
  SRAM_ADDR        <= (others => '-');
  ram_rd_word      <= (others => '-');
  SRAM_DQ          <= (others => 'Z');
  SRAM_LB_N        <= '1';
  SRAM_UB_N        <= '1';

  if (mem_dir_rd = '1') then
   SRAM_ADDR        <= rd_addr(SRAM_ADDR'range);
```

```
   if (rd_buf_idx = '0') then
    SRAM_LB_N   <= '0';
    ram_rd_word <= SRAM_DQ(7 downto 0);
   else
    SRAM_UB_N   <= '0';
    ram_rd_word <= SRAM_DQ(15 downto 8);
   end if;

  elsif (mem_dir_wr = '1') then
   SRAM_ADDR       <= wr_addr(SRAM_ADDR'range);
   SRAM_LB_N       <= wr_buf_idx;
   SRAM_UB_N       <= not(wr_buf_idx);
   SRAM_DQ         <= encoded_pixel & encoded_pixel;
  end if;

 end process;


 ram_regs : process(CLOCK, RESET_N)
 begin
  if (RESET_N = '0') then

   latched_ram  <= (others => '0');
   ram_state    <= IDLE;

  elsif (rising_edge(CLOCK)) then

   if (latch_ram_rd = '1') then
    latched_ram <= ram_rd_word;
   end if;

   ram_state <= next_ram_state;

  end if;
 end process;


end architecture;
```

4 Conclusioni

4.1 Risultati

Il gioco è stato totalmente realizzato in linguaggio VHDL e la gestione sequenziale interamente con macchine a stati finiti, non si è utilizzato un processore. Lo sviluppo del progetto ha evidenziato come sia possibile realizzare un gioco, di media complessità, come *mastermind* utilizzando esclusivamente un linguaggio hardware; inoltre è stato possibile comprendere l'uso di alcuni protocolli per la gestione delle periferiche come la VGA.

Il gioco è quindi un ottimo esempio di cosa è possibile gestire con tecnologie FPGA come la scheda Altera DE1, nonché un esempio di portabilità essendo possibile, con piccoli accorgimenti, trasferire il codice su qualsiasi altra apparecchiatura programmabile dotata dei necessari componenti.

Osserviamo infine che, dalla sequenza di operazioni logiche messe in atto per implementare il gioco mastermind, è possibile generare diverse tipologie di giochi logici, dal gioco del mulino a forza quattro, modificando l'algoritmo di gioco e i componenti mostrati a video.

4.2 Miglioramenti

- Aggiunta della periferica tastiera PS/2 per limitare l'uso dei bottoni della scheda sostituendoli con quelli della tastiera, più intuitivi, fornendo la possibilità di spostarsi da un quadrato all'altro da sinistra verso destra e viceversa, a differenza dell'attuale versione in cui si trasla esclusivamente dal quadrato attuale a quello successivo alla destra e raggiunto l'ultimo quadrato si trasla nel primo.
- Aggiunta di un componente audio che renda più coinvolgente il gioco con suoni per la conferma, la variazione di colore, la vittoria e la sconfitta.
- Aggiunta di ulteriori colori per aumentare la difficoltà nell'individuazione dell'esatta sequenza.