Università degli Studi di Trieste·

Corso di Laurea Triennale in Ingegneria dell'Informazione

Curriculum Biomedica·

Relazione del progetto di fine corso.

Elettronica II - FPGA

Docente: *Marsi Stefano.*

De Micco Dario - 83600219

Martelossi Marta – 83600212

SOMMARIO.

INTRODUZIONE.

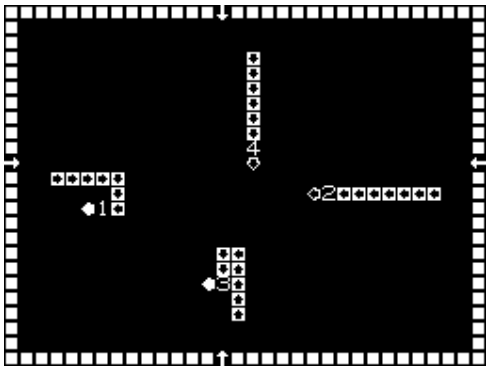
Lo scopo del progetto consiste nella realizzazione del videogioco “Snake” utilizzando la scheda ALTERA DE1 ed in particolare l’FPGA Cyclone II montata su di essa. Oltre all’FPGA vengono sfruttate alcune periferiche del dispositivo come porta VGA, switches, push bottons, porta PS2 e uscita audio.

Per l’implementazione del gioco si è scelto di programmare l’intera logica dell’FPGA in linguaggio Verilog HDL, sfruttando quindi le potenzialità di una programmazione di tipo “hardware”, e di realizzare la logica di processo, il motore di gioco e la gestione grafica, affidandosi a delle macchine a stati finiti.

UN PO’ DI SNAKE…

Le origini del gioco “Snake” risalgono agli anni settanta e non nè esiste una versione definitiva.

La fonte di ispirazione di questo celebre gioco e di tutte le sue varianti è sicuramente il videogioco arcade di azione a labirinto *Barricade*, sviluppato da RamTeK nel 1976, nel quale i giocatori devono muovere i loro blocchi sullo schermo nelle quattro direzioni cardinali, senza sbattere contro il muro che circonda il campo di gioco o gli altri giocatori, e cercando contemporaneamente di intrappolare quest'ultimi.



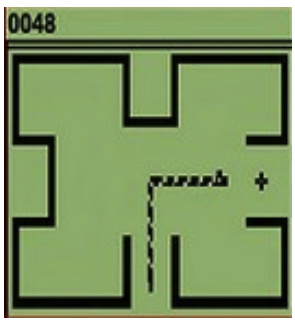
Videogioco Barricade.

Per la sua semplicità, dagli anni ’70 Snake è stato prodotto in numerose piattaforme e varianti, fino a ritrovare nuova fama negli anni novanta grazie ai cellulari, in particolare i Nokia i quali lo possedevano come gioco precaricato.

Il giocatore controlla lo “Snake” facendolo muovere nelle 4 direzioni cardinali.

All’interno del campo di gioco appaiono ad intervalli regolari, vari tipi di frutti o animali rappresentanti il cibo dello snake.

Ogni volta che il serpente “mangia” ciò che appare sul display si allunga, e il giocatore guadagna dei punti.



Videogioco Snake - 2000

L’obbiettivo del gioco è guadagnare quanti più punti possibili evitando di andare a sbattere contro gli ostacoli (come il bordo del campo di gioco o le pareti di possibili labirinti) ma soprattutto contro sé stesso, cosa sempre più difficile man mano

che il corpo si allunga.

Nelle versioni più celebri non solo aumenta la lunghezza dello snake, ma anche la sua velocità rendendo il gioco ancora più entusiasmante.

STRUMENTI UTILIZZATI.

Strumenti Hardware.

1. **Altera Board DE1**
   - Cyclone II EP2C20F484C7N
   - Porta video VGA
   - Porta PS/2
   - Memoria SRAM 512KByte
   - Push Buttons
   - Toggle Switches
   - Uscita audio
   - Memoria Flash 4-MByte
2. **Monitor VGA**
3. **Tastiera PS/2**
4. **Riproduttore audio**

Strumenti Software.

1. **Suite Quartus II 9.1 sp2 Web Edition**
   - Editor testuale
   - Editor di file di memoria mif
   - MegaWizard Plug-In Manager
   - Analizzatore logico SignalTap II
   - Tool di compilazione
   - Programmer JTAG
2. **Photoshop**
3. **Audacity**
4. **Hex Editor**

Per una migliore comprensione della metodologia di realizzazione del gioco, riportiamo di seguito una breve descrizione del funzionamento dei principali componenti hardware utilizzati nello

sviluppo del progetto.

PORTA VGA.

VGA, dall'inglese Video Graphics Array, è uno standard analogico relativo a display per computer introdotto sul mercato nel 1987 da IBM.

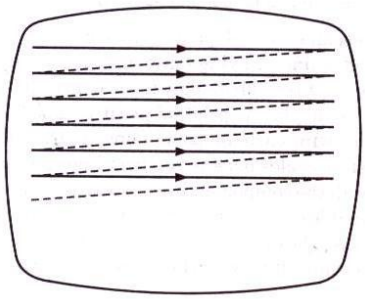
Esistono varie "risoluzioni VGA" (640x480 - 800x600 – 1024x768 - ….) con diverse frequenze di quadro (50 Hz - 60 Hz - 75 Hz - ….) e con possibilità di avere segnale interlacciato, in cui la VGA manda prima le righe pari e poi le righe dispari, oppure segnale progressivo dove vengono mandate tutte quante le righe di seguito.

Per sviluppare il progetto si è deciso di utilizzare lo standard VGA base con risoluzione 640x480 e frequenza di refresh di 60Hz.

I dati (tre tensioni analogiche, una per colore, rappresentanti lo stato del pixel corrente) vengono forniti continuamente in sequenza secondo un "pixel clock" assieme a due segnali di sincronismo (HS = sincronismo di riga e VS = sincronismo di quadro). Di conseguenza il principio di funzionamento della VGA si baserà su due contatori:

1. **CONTATORE DI PIXEL** che conta alla frequenza di clock richiesta (nel nostro caso a 25MHz per un totale di 31.5kHz per riga) i pixel di una riga. Al termine di ogni riga viene generato un impulso di sincronizzazione orizzontale indicante l'inizio di una nuova riga;
2. **CONTATORE DI RIGHE** – conosciuto anche come contatore di frame che alla frequenza di aggiornamento (nel nostro caso a 60Hz) viene azzerato e incrementato ogni volta che il contatore di pixel raggiunge la fine, ed è usato per la sincronizzazione verticale o di quadro.

Potrebbe non risultare subito chiaro il perchè dell'impiego di una simile struttura ma bisogna tenere presente che quando fu introdotto lo standard VGA, la maggior parte dei display funzionava grazie al tubo a raggi catodici, dove un fascio funge da "penna" per ‘‘disegnare’’ su schermo fosforescente, partendo dall'angolo in alto a sinistra, scorrendo verso destra fino a raggiungere l'estremità, per poi ritornare all'estremità sinistra e disegnare la riga successiva.



Percorso fascio elettronico nel CRT.

Inoltre i dati inviati allo schermo compongono un quadro in cui, attorno all'immagine visibile di 640x480 punti, vi è una "area nera", nata per permettere allo schermo di sincronizzarsi e posizionarsi correttamente.

PORTA PS2.

Per comprendere il funzionamento, a grandi linee, della porta PS/2 iniziamo con il riportare una

breve descrizione dell'interfaccia utilizzata ovvero una tastiera PS/2 con la quale il giocatore controlla la direzione dello snake.

La tastiera non è altro che una matrice di tasti, monitorizzati da un processore interno. Nonostante i processori siano diversi e varino in base alle caratteristiche della singola tastiera, tutti svolgono lo stesso compito: controllare quale tasto è stato premuto o rilasciato e spedire all'host un dato appropriato.

Inoltre i processori si occupano di mettere in uno stato di idle e, quindi di bufferizzare, i dati nel caso in cui la linea di comunicazione risulti occupata.

Si fa notare che, siccome tramite il protocollo PS2 è possibile gestire l'I/O da tastiera con molti dispositivi, per mantenere maggiore generalità in questo contesto ci si è riferiti alla scheda DE1, con il generico termine di host.

Il pacchetto di informazioni che il processore interno alla tastiera trasmette all'host quando vede che un tasto è stato premuto, rilasciato o tenuto premuto viene chiamato SCAN CODE e può essere di due tipi:

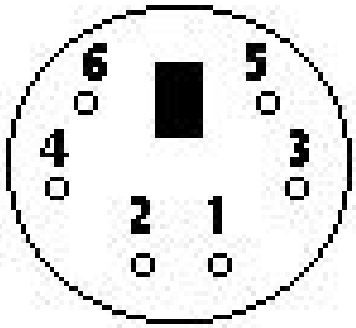
1. **MAKE CODE:** spedito quando un tasto è premuto o viene tenuto premuto.
2. **BRAKE CODE:** utilizzato quando un tasto è rilasciato.

Ogni tasto viene associato ad un unica coppia < make code / break code >, con la quale l'host può determinare esattamente cos'è successo e su quale tasto.

È importante tenere sempre presente che i codici inviati identificano univocamente un tasto sulla tastiera e non un carattere: ovvero non è definita alcuna relazione tra scan code e codifica ASCII (tradurre scan codes in codici ASCII è compito dell'host).

Compreso ora, almeno in linea generale, il funzionamento dell'interfaccia possiamo passare alla descrizione del protocollo PS2 vero e proprio per la comunicazione tra tastiera e board.

Il protocollo PS/2 è un protocollo seriale, sincrono e bidirezionale sviluppato dall'IBM, con connettore così composto:

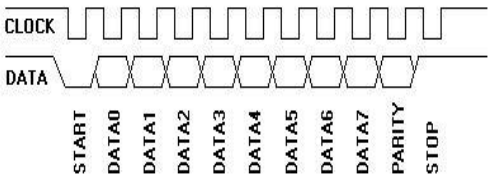


Mini-DIN a 6 piedini:
1) Data,
2) Non implementato,
3) Massa,
4) Vcc +5V,
5) Clock,
6) Non implementato

Dalla figura si intuisce che tutta la comunicazione tra host e device e, viceversa, si basa su un bus formato da due linee bidirezionali: la linea di clock e la linea dati caratterizzate da due stati logici differenti, uno alto e uno basso.
Quando entrambe le linee sono in uno stato logico alto, il bus è in attesa e, questo, è l'unico stato durante il quale è permessa la trasmissione dei dati da parte della tastiera.
La board DE1 ha il controllo finale sul bus e può in ogni momento inibire la comunicazione mettendo a massa la linea di clock, mentre è la tastiera a generare il quest'ultimo.
I dati spediti dalla tastiera alla board sono letti sul fronte di discesa del clock e, viceversa, i dati trasmessi dalla board alla tastiera sono letti sul fronte di salita del clock.
Per quanto riguarda la realizzazione del gioco ci si è occupati esclusivamente del flusso di dati dalla tastiera alla scheda che avviene nel seguente modo:

- Il device, nel nostro caso la tastiera, controlla che il bus sia libero da almeno 50 us.
- Se il bus non è libero la board sta inibendo la comunicazione e la tastiera bufferizza i dati da spedire.
- I dati vengono scritti dalla tastiera quando il clock è alto.
- Tutti i dati vengono trasmessi in modo seriale, ovvero viene trasmesso un bit alla volta, e ogni byte viene inserito in un frame di 11 bit, come da figura a lato.
- Il dato viene letto dall'host sul fronte di discesa del clock.

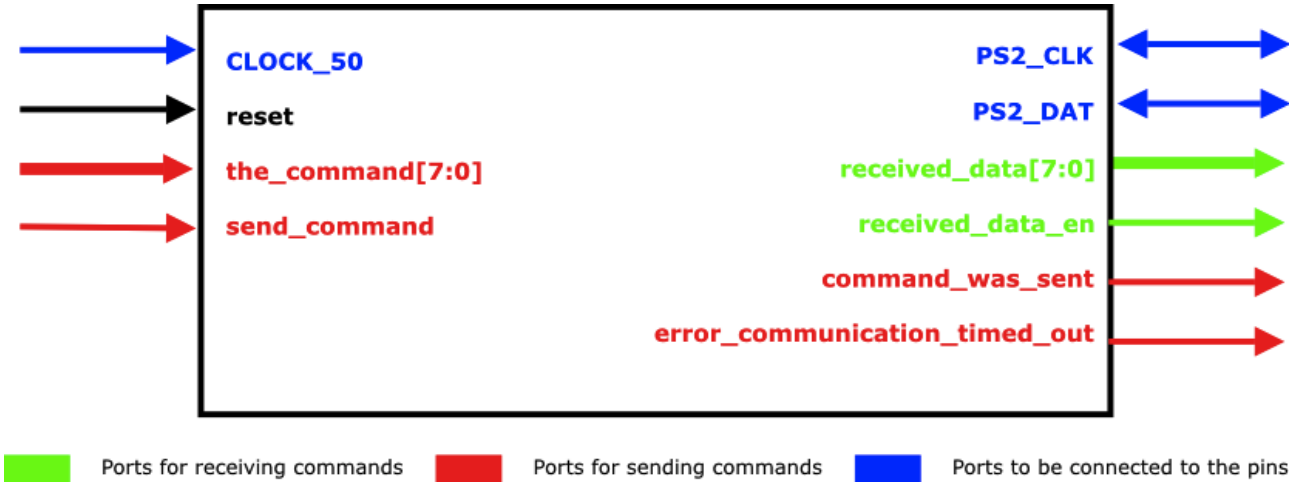
Trasmissione seriale dati.



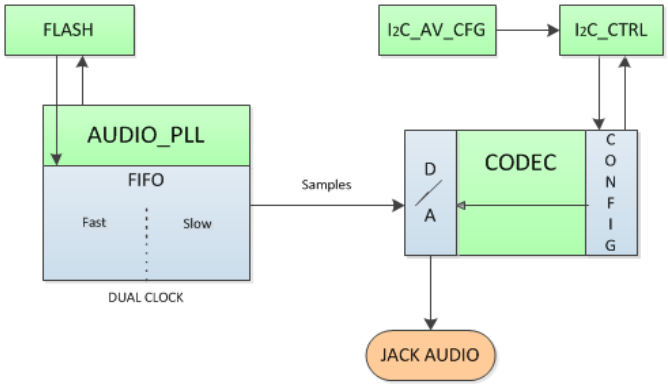
1 bit: START (sempre 0)
8 bit: DATI (LSB primo)
1 bit: PARITY BIT
1 bit: STOP(sempre 1)

Ovviamente i dati trasmessi dalla tastiera corrispondono allo scan code dei tasti che vengono premuti.

Per completezza si riporta infine lo schematico della porta PS2.



BLOCCO AUDIO.



Grazie allo schema a blocchi riportato a lato, è possibile dare una prima, generale spiegazione del funzionamento della parte audio del progetto.

I dispositivi fisici interessati sono:

1. La **MEMORIA FLASH**, integrata nella DE1, utilizzata per memorizzare il file audio. Si tratta di una memoria SPANSION modello S29AL032D, a 32Mbit alimentata a 3 volt, basata su CMOS, accessibile come una normale memoria asincrona statica parallela.

   Per ulteriori informazioni e dettagli, come lo schema logico, rimandiamo ai datasheet o alla documentazione Altera in dotazione con la board DE1.

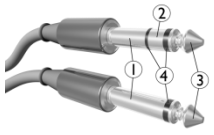
2. Il **CODEC AUDIO**, modello WOLFSON WM8731 presente anch'esso nella DE1, che permette di gestire flussi di dati audio e svolgere su di essi operazioni quali regolazione del volume, modifica delle frequenze di campionamento e gestione dell'uscita audio analogica utilizzata. Dai datasheet si ottengono le configurazioni dei vari registri che compongono il codec, in modo da poterlo impostare secondo le specifiche volute.

   Di seguito riportiamo i registri utilizzati.

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000111 Digital Audio Interface Format	1:0	FORMAT[1:0]	10	Audio Data Format Select 11 = DSP Mode, frame sync + 2 data packed words 10 = I²S Format, MSB-First left-1 justified 01 = MSB-First, left justified 00 = MSB-First, right justified
	3:2	IWL[1:0]	10	Input Audio Data Bit Length Select 11 = 32 bits 10 = 24 bits 01 = 20 bits 00 = 16 bits
	4	LRP	0	DACLRC phase control (in left, right or I²S modes) 1 = Right Channel DAC data when DACLRC high 0 = Right Channel DAC data when DACLRC low (opposite phasing in I²S mode) or DSP mode A/B select (in DSP mode only) 1 = MSB is available on 2nd BCLK rising edge after DACLRC rising edge 0 = MSB is available on 1st BCLK rising edge after DACLRC rising edge
	5	LRSWAP	0	DAC Left Right Clock Swap 1 = Right Channel DAC Data Left 0 = Right Channel DAC Data Right
	6	MS	0	Master Slave Mode Control 1 = Enable Master Mode 0 = Enable Slave Mode
	7	BCLKINV	0	Bit Clock Invert 1 = Invert BCLK 0 = Don't invert BCLK

0000100 Analogue Audio Path Control	0	MICBOOST	0	Microphone Input Level Boost 1 = Enable Boost 0 = Disable Boost
	1	MUTEMIC	1	Mic Input Mute to ADC 1 = Enable Mute 0 = Disable Mute
	2	INSEL	0	Microphone/Line Input Select to ADC 1 = Microphone Input Select to ADC 0 = Line Input Select to ADC
	3	BYPASS	1	Bypass Switch 1 = Enable Bypass 0 = Disable Bypass
	4	DACSEL	0	DAC Select 1 =Select DAC 0 = Don't select DAC
	5	SIDETONE	0	Side Tone Switch 1 = Enable Side Tone 0 = Disable Side Tone
	7:6	SIDEATT[1:0]	00	Side Tone Attenuation 11 = -15dB 10 = -12dB 01 = -9dB 00 = -6dB

Connettore TRS.



3. L'**USCITA JACK** (sulla board, in verde) al quale andrà collegato il dispositivo di riproduzione audio (casse).

   La connessione avviene tramite connettore TRS, la cui sigla deriva da TIP ovvero punta (in figura il numero 3), dedicato al canale sinistro, o ad entrambi i canali

se in modalità mono, RING, anello (numero 2), utilizzato in modalità stereo per il canale destro, e infine SLEEVE, ovvero manica (numero 3), collegato a massa.

I rimanenti blocchi dello schema iniziale, sono composti da codice compilato, il cui funzionamento verrà descritto in dettaglio nelle pagine seguenti.

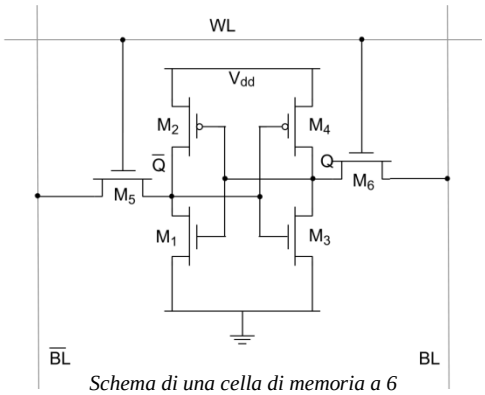
MEMORIA SRAM.

La memoria SRAM, acronimo di Static Random Access Memory, è una memoria RAM volatile, che memorizza quindi i dati solo fino a quando è collegata all'alimentazione e che non necessita di refresh.

L'architettura di base di una RAM statica include uno o più 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 della board DE1 è organizzata in celle di memoria contenti 6 transistor CMOS (tipicamente una cella di memoria è costituita da 6 transistor ma esistono anche celle a 4 transistor).

Il circuito contiene un flip-flop SR bistabile (4 transistor) e due porte di trasmissione che collegano la cella alle linee dei dati.



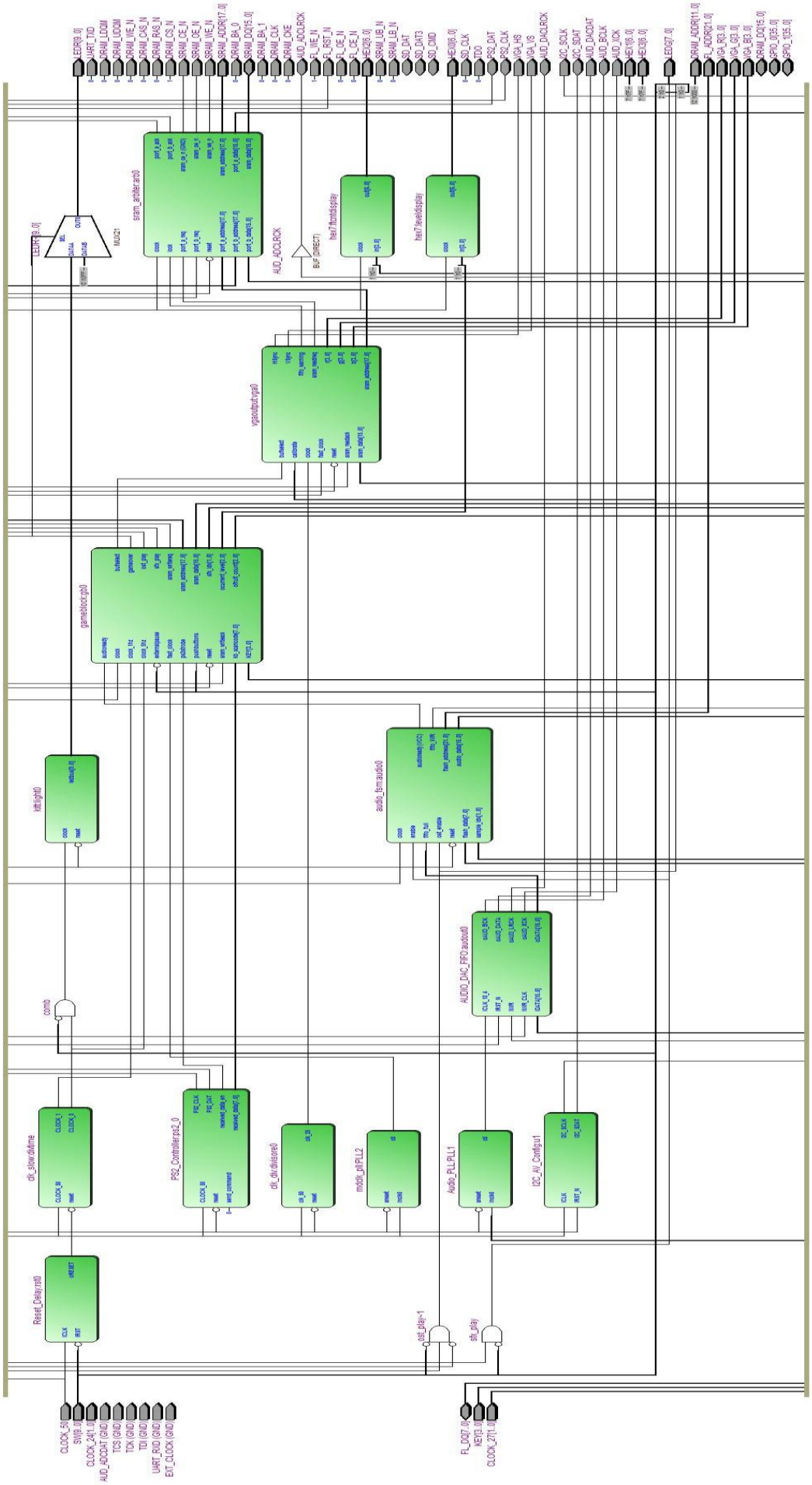
Schema di una cella di memoria a 6 transistor

La SRAM in questione (modello IS61LV25616) 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 (Chip Enable, Output Enable, Write Enable, Lower Byte, Upper Byte). Oltre a questi sono presenti naturalmente i piedini di alimentazione e di massa.

STRUTTURA DEL PROGETTO.

L'intero progetto è schematizzato da diversi blocchi funzionali comunicanti tra loro, partendo dai più semplici divisori di clock, per poter adattare il ritmo di gioco, ai più complessi dedicati alla gestione vera e propria del gioco o della porta porta vga.

Riportiamo nella pagina seguente l'elenco dei blocchi necessari alla creazione del gioco e l'intero schema a blocchi.



BLOCCHI:

- Reset_Delay
- *clk_slow*
- *clk_div*
- *rndclk_pll*
- *Audio_PLL*
- *I2C_AV_Config*
- *AUDIO_DAC_FIFO*
- *audio_fsm*
- *gameblock*
- *vgaoutput*
- *sram_arbiter*
- *hex7:ftcntdisplay*
- *hex7:leveldisplay*

Facciamo notare che alcuni moduli di proprietà dell'Altera, sono stati rielaborati in piccole parti per poterli adattare alle specifiche del progetto. Sono stati utilizzati anche moduli, in parte rielaborati, di vecchi progetti come Tetris o la sveglia. Passiamo quindi ora alla descrizione più dettagliata di ogni singolo modulo.

SNAKE.

Aprendo Quartus è possibile visualizzare il modulo "snake" non presente nello schema a blocchi riportato in precedenza. Si tratta di un modulo generale che si occupa di istanziare e relazionare tra loro gli altri moduli che compongono il gioco. Lavora quindi "in sinergia" con l'altro blocco principale, il "gameblock", di cui sarà data successivamente una descrizione.

RESET_DELAY.

Il modulo Reset_Delay ha il compito di generare un breve ritardo prima della comparsa della schermata di gioco e dell'inizio del gioco vero e proprio. Ciò permettere al giocatore di prepararsi prima di una nuova partita.
In esso è anche implementata una funzione di reset, attuabile mediante lo switch SW9 (posizione up seguito da posizione down).

CLK_SLOW, CLK_DIV.

La scheda DE1 utilizza un clock fisso a 50Mhz generato da un oscillatore al quarzo. Sfruttando l'oscillatore è stato possibile temporizzare tutti i vari blocchi del progetto.

1. "**CLOCK_SLOW**" è stato creato dividendo il clock della DE1 fino ad ottenere 1Hz ed è utilizzato per la temporizzazione del timer in alto.
   Da questo blocco si ottiene in output un ulteriore clock da 5Hz che viene impiegato nel modulo kitt dei led rossi.
2. "**CLOCK_DIV**" è un altro divisore di clock per adattare la frequenza della DE1 alla frequenza della porta VGA che lavora a 25MHz. Per ottenerlo è stato semplicemente diviso per 2 il clock della DE1.

AUDIO_PLL - Modulo Altera.

Questo modulo implementa un PLL per generare la frequenza di "master clock" necessaria al funzionamento del CODEC.

I2C_AV_CONFIG – Modulo Altera.

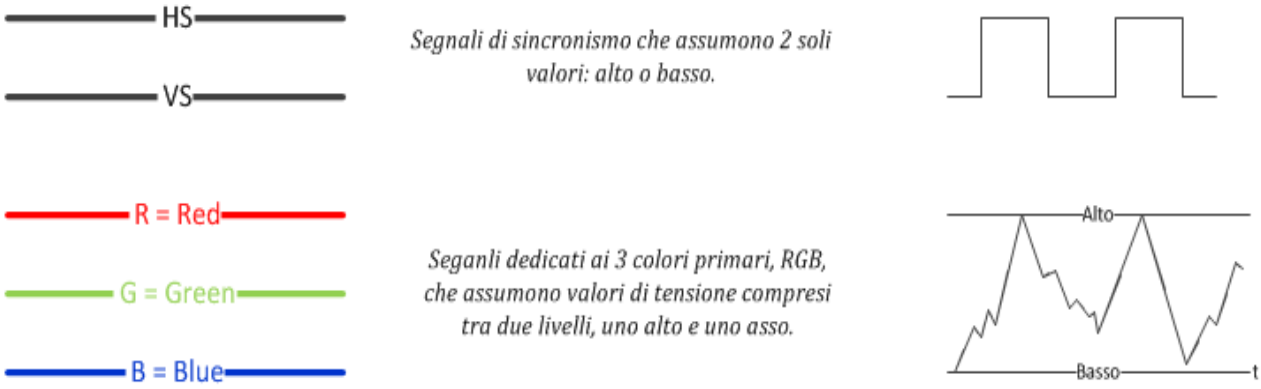
Contiene e gestisce la comunicazione con il convertitore audio D/A presente nella DE1.
In esso sono stati impostati opportunamente i valori che andranno poi ad essere memorizzati nei registri di configurazione del convertitore mediante una comunicazione (gestita dal modulo I2C_CONTROLLER) secondo il protocollo I2C. Si è agito sui valori di questi registri (descritti nel datasheet del CODEC) per cambiare la frequenza di campionamento.

AUDIO_DAC_FIFO. – Modulo Altera.

In questo blocco sono contenuti i parametri per la gestione fisica della comunicazione verso il convertitore. I dati, trasmessi ad esso in via seriale, vengono bufferizzati da una memoria FIFO a clock separati: il convertitore preleva i campioni con una cadenza legata alla frequenza di campionamento, mentre la memoria viene riempita con i suddetti campioni ad una velocità indipendente (e superiore) dalla prima.
Il modulo si occupa, inoltre, di generare i segnali di temporizzazione per il convertitore a partire dal "master clock".
Il funzionamento dell'intero codec è descritto nel datasheet "Wolfson-WM8731".

VGA OUTPUT.

Il segnale al monitor è prettamente analogico e arriva grazie a 5 linee: 2 dedicate ai sincronismi e 3 ai colori primari.



Il modulo VGA_OUTPUT ha quindi il compito di gestire tutti e 5 questi segnali.
Come già accennato in precedenza, i sincronismi sono fondamentali per la gestione delle immagini sull'output video in quanto essi comandano il pennello elettronico (fascio) scandendo, pixel per pixel, il quadro intero.

- I 640 pixel disegnabili orizzontalmente sono compresi tra due zone di front porch e back porch, che non vengono però disegnate, rappresentando di fatto due bande nere: tutte le zone sono identificate dal segnale di sincronismo orizzontale posto a livello logico alto.

  Per far tornare il pennello elettronico a capo e permettergli così di iniziare a scandire una nuova riga, viene fatto assumere al sincronismo orizzontale, livello logico basso per un breve lasso di tempo.
- Il disegno delle 480 righe è, invece, regolato dal segnale di sincronismo verticale che agisce, sul pennello elettronico, in maniera analoga al sincronismo orizzontale, segnalando però, invece della fine e dell'inizio riga, la fine di ogni frame e l'inizio del successivo.

Per una migliore comprensione si riportano le tabelle con le specifiche dei due sincronismi analizzati.

Sincronismo orizzontale (di riga).

Pixel (tot.800)	96 (sync pulse)	48 (back porch)	640 (area visibile)	16 (front porch)
Durata (tot.31.77us)	3.81 us	1.91 us	25.42 us	0.63 us

Sincronismo verticale (di quadro).

Linee (tot.525)	2 (sync pulse)	33 (back porch)	480 (visible area)	10 (front porch)
Durata (tot.16.68 ms)	0.06 ms	1.05 ms	15.25 ms	0.32 ms

Per quanto riguarda i tre segnali dedicati ai colori RGB, le loro combinazioni forniscono le colorazioni dei pixel sullo schermo. In base alla profondità di ciascun colore fondamentale, rispettivamente rosso, verde e blu, si ottengono tutte le possibili colorazioni.
La scheda DE1 contiene un DAC a 4bit per canale (16 livelli per colore) in uscita realizzando così 16x16x16 = 4096 possibili colori distinti.
Tutto ciò che viene visualizzato sul monitor è gestito completamente dal frame buffer (memoria buffer della scheda video).
Un problema comune a tutte le interfacce video è che, se il componente video (nel nostro caso il modulo VGA_OUTPUT che invia i segnali allo schermo) non riceve esattamente tutti i dati di cui ha bisogno nel momento in cui né ha bisogno, l'immagine visualizzata appare incompleta, distorta o completamente instabile. Il compito del frame buffer è quindi proprio quello di non permettere il presentarsi di questo problema. Per farlo memorizza su SRAM un intero fotogramma (frame) sullo

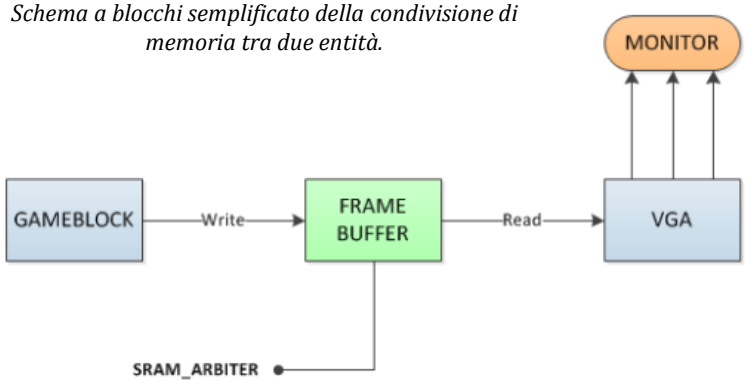
schermo, in modo tale da permettere al sistema grafico di avere sempre l'immediata disponibilità dei dati. Avendo a disposizione un intero frame di dati pronto all'uso, l'arbitro ci assicura che la richiesta sia soddisfatta alla frequenza in cui il sistema grafico richiede i dati.
Per rendere il frame buffer ancor più robusto (si vedano i meccanismi implementati nel modulo dell'arbitro per impedire la mancanza di dati al modulo VGA_OUTPUT), si è implementata la gestione di **una piccola memoria** di tipo **FIFO** (First In First Out). Questa funge da deposito dati temporaneo, diventando quindi una sorta di buffer ovvero un ‘‘tampone’’ per poter immagazzinare un certa quantità di pixel. Ciò permette di non dover richiedere, e, di conseguenza, di aver sempre pronto, un pixel dalla SRAM per ogni pixel che deve essere inviato all'uscita VGA.
Quando viene eseguita un'operazione di lettura o scrittura gli appositi segnali indicano lo stato del buffer. I dati prelevati dalla SRAM vengono copiati nella FIFO fino a che essa non sia piena ‘‘abbastanza’’ (fifo_writefull). Contemporaneamente vengono prelevati dei dati alla cadenza richiesta dalla VGA che svuotano la memoria FIFO. Subito il livello della FIFO comincia a calare e si provvede a richiedere un nuovo dato dalla SRAM (essendo la porta "verso la VGA" prioritaria sulla SRAM, l'arbitro concede la lettura non appena viene terminata l'operazione in corso).
Nel caso la FIFO si svuoti molto velocemente, e si trovi troppo vicino al livello vuoto, viene asserito un segnale (fifo_almostempty) che obbliga l'arbitro a servire solo le richieste per la VGA, agendo sul segnale di ‘‘lock’’, in modo che la FIFO possa essere riempita.
Essendo il pixel clock di 25MHz, per non dover gestire problemi vari abbiamo reso disponibile un pixel ogni 40ns (1/25MHz).
Il modulo Verilog che effettivamente genera la FIFO (di dimensione 1024x4pixel = 4096pixel) per la VGA è "FIFO_VGA", generato dal MegaWizard Plug-In Manager.
Nonostante non sia stato utilizzato nella realizzazione del progetto (riusciamo a scrivere tutto in tempo utile), è stato comunque predisposto un sistema di double-buffering nel caso in cui si abbiano molti dati e non si riesca a scrivere tutto nella durata dei "tempi morti": ciò permette di avere in SRAM due copie intere dell'immagine. Una, letta e mantenuta sempre uguale, che viene inviata allo schermo, mentre l'altra (accessibile in scrittura) è modificabile senza rischio che si vedano "comparire" le modifiche ovvero che il tempo di modifica risulti >> del tempo di visualizzazione del fotogramma. Una volta completata la modifica si invertono i ruoli: su schermo viene inviata la copia appena modificata mentre si può agire su quella visualizzata fino ad un attimo prima.

SRAM_ARBITER.

La memoria RAM utilizzata, raffigurata nello schema dal frame buffer, è una memoria a porta singola e ciò vuole dire che può svolgere una sola operazione alla volta, o di lettura o di scrittura.

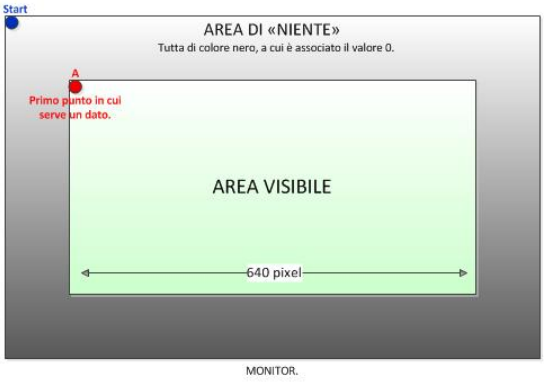
Per accedervi è necessario impostare un indirizzo sull'apposito bus indirizzi ed i vari segnali di controllo (Chip Enable, Output Enable, Write Enable) in base a ciò che si vuol fare.
Durante un'operazione di lettura, la memoria pone il contenuto dell'indirizzo impostato sul bus dati (che verrà letto), mentre in un'operazione di scrittura la memoria preleva il contenuto posto sul bus dati e lo pone nella cella indirizzata.
Ora, a causa proprio della presenza della porta singola, se due entità devono condividere la stessa memoria, è necessario introdurre un **ARBITRO** (SRAM_Arbiter), in quanto, potrebbe accadere che le due entità cerchino di accedere a diversi indirizzi in momenti coincidenti.
Due output non possono essere attaccati insieme allo stesso filo (p.e. l'input del bus indirizzi della SRAM) a meno che non ci sia un sistema che scolleghi uno o l'altro a seconda delle necessità. Se i due output cercassero di porre valori diversi (questi valori sono bit, quindi diversità fra i numeri 0 – connesso a massa e 1 – connesso al positivo dell'alimentazione) si rischierebbe di fare un cortocircuito e comunque il sistema non funzionerebbe correttamente non sapendo quale dei due dati trattare. L'arbitro funge quindi da interfaccia per entrambi i sistemi e garantisce l'accesso alla memoria, uno per volta, a ciascun sistema evitando così collisioni.
Quando giunge una sola richiesta l'arbitro nè verifica la validità e in caso affermativo la soddisfa collegando la SRAM all' entità che ha richiesto l'accesso nel momento in cui lo ha fatto.
Nel caso in cui entrambe le entità richiedano l'accesso contemporaneamente, l'arbitro si occupa di gestire le priorità e mettere in attesa l'entità non ancora servita.
Come si può facilmente vedere dallo schema riportato, nel modulo "SRAM_ARBITER" le due entità sono il modulo precedentemente descritto VGA (port_a di sola lettura) che si occupa di leggere l'immagine (frame) salvata in memoria (buffer) e porre i segnali giusti sull'uscita VGA analogica e, il modulo del gioco, "GAMEBLOCK", (port_b di sola scrittura) che invece si occupa di scrivere l'immagine in memoria ogni volta che deve essere apportato un cambiamento a ciò che è visualizzato sullo schermo.

Schema a blocchi semplificato della condivisione di memoria tra due entità.



La porta A ha la priorità sulla porta B perché il modulo VGA deve sempre ricevere il segnale: per questo motivo è stato previsto un sistema di ''locking'' (ovvero di blocco) affinché la porta prioritaria possa, in caso di necessità di più dati in maniera continuativa, assicurarsi l'accesso alla memoria SRAM.

Riassumendo: l'idea generale è che una delle due porte possa fare una richiesta e che l'arbitro decida se questa possa venire servita oppure no. Se avviene una richiesta in contemporanea su entrambe le porte viene servita quella con priorità superiore.
Prima di poter servire una nuova richiesta è, però, necessario terminare tutto ciò che si è iniziato: quindi, se si sta servendo una richiesta da B, A dovrà comunque aspettare fino al suo completamento (motivo che ha portato all'aggiunta di un buffer FIFO nel modulo VGA).
Poiché la scrittura (porta B) richiede più cicli di clock per essere completata, se la porta A necessita di dati in maniera continuativa, può fare una richiesta di "lock", cioè mantenere l'accesso fino a quando non sarà terminata la condizione di necessità di dati. Questo impedisce che ci sia mancanza di dati alla porta A dovuta al fatto che si sta terminando una scrittura dalla porta B.



Si fa notare che la VGA legge la RAM (640 pixel in sequenza, a 25MHz) solo quando è all'interno dell'area visibile, mentre quando si trova nell' area di "niente" ovvero in area non visibile (inizio riga e fine riga), essa non richiede di inviare dati di pixel (RGB).
Nell'area non visibile ci sono quindi dei tempi che vengono sfruttati per poter scrivere la RAM, non essendo occupata in lettura dalla VGA. Durante questi momenti non vengono letti dati dalla FIFO che, una volta piena, smette di chiedere dati alla SRAM: possono quindi essere eseguite tutte le scritture pendenti.

GAMEBLOCK.

È il cuore dell'intero progetto. In questo modulo, infatti, è contenuto il gioco vero e proprio che, gestendo gli input e gli output, si occupa di mettere in comunicazione le varie risorse descritte fin ora. Il blocco è composto interamente da codice, in quanto svolge soltanto una funzione di tramite tra le varie risorse fisiche: gestisce gli stimoli in ingresso (i comandi dell'utente) e, in base alle condizioni attuali, determina l'evoluzione del gioco, visibile su schermo, agendo quindi direttamente sul framebuffer e producendo l'immagine visualizzata.
Andiamo ora ad analizzarne gli aspetti fondamentali.

Un **FRAME** è un'immagine definita come una matrice X×Y = 640×480, delle stesse dimensioni dell'area visibile della VGA (640×480), in cui ogni valore a 4 bit rappresenta un pixel (16 colori).

Il frame viene salvato nella SRAM e una copia dell'immagine occupa (640×480)/4 ossia 76800 indirizzi della SRAM, in quanto questa contiene un bus dati largo 16 bit e quindi, accedendo a 16 bit, si accede in realtà a 4 pixel da 4 bit ciascuno.
La velocità di scrittura dei dati sulla SRAM è di 1 parola (16 bit/4 pixel) ogni due colpi di clock.
Per la generazione della grafica di base, che comprende tabella di gioco (di dimensione 10×18), la frutta da far comparire, il tempo e il punteggio, il codice ''passa'' tutta l'immagine mediante due contatori: 1 - contatore orizzontale X

2 – contatore verticale Y.

Questi contatori seguono lo stesso principio di quelli utilizzati per i sincronismi orizzontali e verticali della VGA, ma a differenza di quest' ultimi, non sono presenti le zone di banking: partendo quindi dall'origine, di coordinate [0;0], procedendo per la prima riga (coordinate [639;0]), passando poi alla seconda ([0,1]) fino alla sua fine ([639,1]) e così avanti fino alla fine del frame ([639,479]), scandiscono tutto il frame.

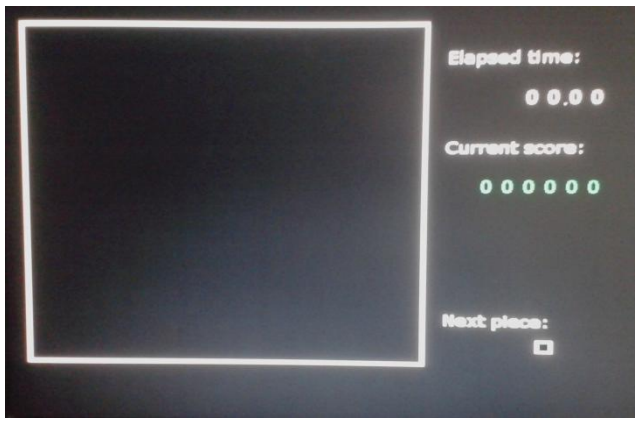


Immagine del primo frame.

Il primo frame (in figura), viene disegnato una sola volta: questo significa che i dati nella SRAM che corrispondono a tali pixel, una volta scritti non vengono più modificati. Vengono però modificati i valori corrispondenti ai pixel che rappresentano le cifre del punteggio, il tempo, l'informazione "Next piece" sul prossimo frutto, e soprattutto vengono riscritti i valori all'interno del quadrato di gioco, che saranno aggiornati parzialmente ad ogni mossa e totalmente ad ogni cambio livello (p.e. quando si passa ad un nuovo livello il campo di gioco si modifica con l'introduzione di labirinti).
Le scritte "Elapsed time:", "Current score:", "Next piece:" e le cifre sono state disegnate con Photoshop, ma è possibile utilizzare qualsiasi altro software di elaborazione immagini che permetta di salvare l'immagine come dati grezzi.

Il **CAMPO DI GIOCO** (tabella di gioco) è un'area formata da 64x64 celle elementari, ciascuna di dimensione 5x5 pixel e spaziate di 1 pixel. Ognuna di queste celle può avere uno dei seguenti tipi: snake, frutta, muro, vuoto.
Per quanto riguarda invece le **MAPPE DI GIOCO** (configurazione del campo a inizio livello)**,** sono state create anch'esse tramite Photoshop, in scala di grigi: si avranno allora 8 bit per pixel,

dove 00 rappresenta il nero, FF il bianco e i valori intermedi identificano le varie tonalità di grigio possibile. La risoluzione è data dalle dimensioni del quadro di gioco.
Una volta create le immagini desiderate otteniamo 8 file RAW contenenti le mappe dei vari livelli. Ogni livello è composto da 64x64 (= 4096) caselline (durante il design processing erano pixel), e ogni casellina è rappresentata da 8 bit. Visto che ogni casellina rappresenta quella che sarà poi una cella del campo di gioco, due soli valori saranno possibili: per praticità (immagine b/n) si sono scelti 00 per assenza di muro e FF per presenza di muro. Sono state così disegnate le "mura dei labirinti" come pixel bianchi su sfondo nero.
A questo punto è necessario concatenare i file e convertirli in binario. Per farlo si è utilizzato il CMD di Windows e il comando COPY:

COPY file1+file2+file3 uscita
Inserisce il file2 ala fine del file1 e poi il file3 alla fine di tutto.

Poiché vogliamo trattare tutto in binario dobbiamo specificare le nostre intenzioni sia alla sorgente che alla destinazione. Si avrà allora:

COPY /B file1+file2+file3 /B uscita
Concatena i file a livello di bit ovvero prende tutti i bit del primo file e ci inserisce in coda tutti i bit del secondo e così via.

Fatto ciò, otteniamo 32768 byte di dati rappresentanti tutte le mappe degli 8 livelli in sequenza.
La concatenazione binaria è fondamentale: senza questo passaggio, infatti, i file verrebbero trattati come semplice testo e verrebbero quindi cercati solo caratteri ASCII.
Per importare infine il file grezzo/binario in Quartus si è passati per un formato intermedio con estensione .hex, convertendo tutto in Intel HEX, sempre a 8 bit.
È possibile a questo punto aprire il file in Quartus e visualizzarne il contenuto che risulta essere formato da "parole" da 8 bit in cui ogni singola "casella" contiene due cifre esadecimali. Ogni parola rappresenta quindi un quadratino della mappa.
Visto che il file viene caricato sull'FPGA su una memoria ROM a sola lettura, cerchiamo di risparmiare un po' di memoria facendo in modo che ogni quadratino contenga un solo solo bit di informazione:
- 0 se il quadratino è vuoto
- 1 se c'è un muro.

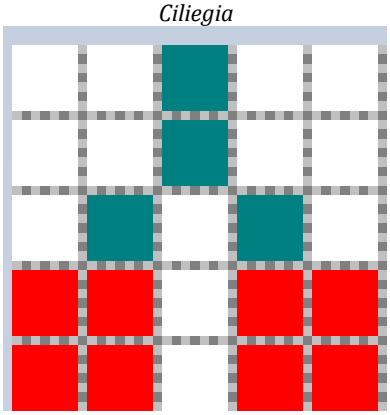
Si è convertito quindi il file in modo che le parole (byte) inizialmente da 8 bit diventassero da 1 bit, eliminando gli altri bit contenenti solo informazioni ridondanti.
A questo punto avremo un file con 32768 (64 w×64 h×8 n, width/height/number of levels) parole da 1 solo bit, che, come già detto, verrà memorizzato in una ROM che sarà poi letta.

Per la creazione delle immagini della **FRUTTA,** si è seguito un procedimento analogo a quello usato per le mappe: tutti gli 8 tipi di frutta inseriti nel gioco, sono stati creati con l'ausilio di Photoshop, convertiti in 16 colori, salvati in RAW, concatenati e trasformati in HEX (con i comandi precedentemente descritti), per essere infine importati in Quartus e salvati nelle ROM implementate nell'FPGA.

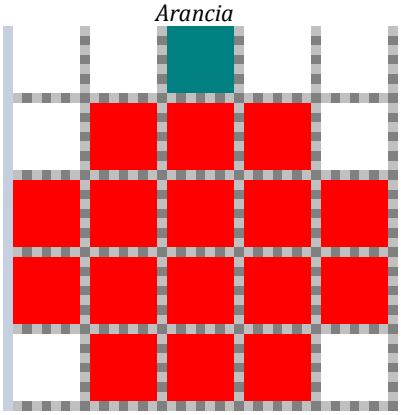
Ogni frutto, che occupa una cella elementare, è composto da 5x5 (=25) pixel. Ogni pixel è rappresentato con un valore da 4 bit, con il quale si rappresenta uno dei 16 colori messi a disposizione dalla palette che verrà descritta in seguito.

Per la modifica di un frutto è necessario quindi modificare ogni singolo blocco da 25 pixel, come mostrato dalla figura sottostante.

Ciliegia



Arancia



A differenza delle mappe, che erano in scala di grigi, la frutta è a 16 colori e, di conseguenza, ogni byte (parola) rappresenta 2 pixel, da 4 bit ciascuno.

La conversione, che nelle mappe aveva portato ad avere un bit di informazione per ogni quadratino, nel caso della frutta è stata effettuata da parole a 8 bit a parole a 4 bit. La differenza quindi rispetto alle mappe è che in questo caso il numero di parole viene raddoppiato e nessun bit viene eliminato.

I frutti vengono disegnati a partire dalle informazioni contenute nel file *pieces_rom.mif*. Aprendo in QuartusII il file, *resources/graphics_roms/pieces_rom.mif*, appare la tabella di figura con la quale viene appunto raffigurata la frutta.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	F	F	6	F	F	F	9	9	.....
08	9	F	9	9	9	9	9	9	.....
10	9	9	9	9	F	9	9	9	.....
18	F	F	F	B	1	F	F	B	.....
20	B	F	F	B	B	F	F	F	.....
28	B	B	F	F	F	F	B	B	.....
30	F	F	F	F	6	F	F	F	.....

Tabella raffigurante la frutta in Quartus II.

Il gioco è gestito tramite diverse **MACCHINE A STATI FINITI**. In particolare due sono le macchine principali:

1. **FSM0** - macchina a stati finiti per la **GESTIONE DELLA GRAFICA** che si occupa dell'aggiornamento del frame buffer con la situazione attuale del gioco (leggendo la tabella di gioco), scrivendo le modifiche relative all'area di gioco nella SRAM e, quando necessario, modifica le informazioni relative al punteggio al tempo e al prossimo frutto.
2. **FSM1** – macchina a stati finiti per la **GESTIONE DEL GIOCO VERO E PROPRIO** (mainFSMstatus): in generale si occupa di aggiornare la tabella di gioco, impostando lo stato opportuno di ogni singolo quadratino, fornendo così informazioni utili alla macchina del punto 1 che si occuperà di disegnare quanto visualizzato a video.

Si è scelto un approccio sequenziale, piuttosto che parallelo, in modo da limitare l'uso di risorse (quindi area occupata nel chip) piuttosto che privilegiare una velocità di esecuzione elevata. Si è visto poi (si vedano le conclusioni) che la scelta si è rivelata ottima in quanto l'uso di risorse è molto basso e non si manifesta alcun collo di bottiglia. Questo, in vista di una possibile commercializzazione di ASIC su larga scala (per esempio da integrare in giochi low-cost) riduce il costo del chip.

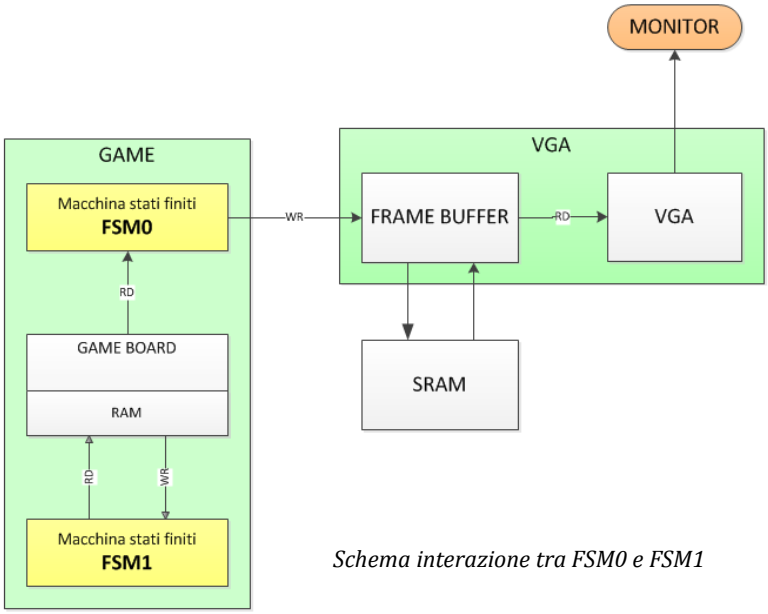
Sempre nell'ottica di ridurre la complessità del chip, si è scelto di utilizzare memorie RAM reali (dell'FPGA in questo caso) al posto di registri dove la mole di dati fosse significativa. Seppur questo riduca la velocità e obblighi in certe situazioni a pensare il codice in maniera sequenziale, il risparmio di risorse risulta notevole e la struttura a macchine a stati si presta perfettamente a svolgere la gestione. Esse introducono anche i ritardi necessari per non violare i tempi di setup e hold delle memorie e permettono una facile migrazione verso RAM esterne qualora si volesse risparmiare ancora di più su un ipotetico ASIC. Analogamente è possibile fare con le ROM (che nell'FPGA vengono comunque caricate nei blocchi RAM): la possibilità di utilizzare di una ROM (anche tipo Flash) esterna per memorizzare non solamente l'audio ma anche le risorse grafiche (scritte, mappe, frutta) permette una facile personalizzazione.

Il gioco è regolato da un "orologio di gioco" (GAME CLOCK) e da una serie di eventi: ad ogni "tick" del "gameclock" inizia l'elaborazione degli eventi sulla base dello stato attuale. Seppure gli eventi (pressione di un tasto per cambio direzione, comparsa di nuova frutta, scomparsa di frutta, etc) possano accadere in qualsiasi momento, essi saranno elaborati e quindi avranno conseguenze sul gioco solo al "gametick" successivo.

A differenza del progetto Tetris, nel quale se venivano schiacciati più velocemente i tasti sulla tastiera (ex: dx-dx-dx) anche il tetramino doveva muoversi più velocemente nella maniera indicata

(la pressione di un tasto portava all'immediato cambiamento nel gioco), nel nostro caso il movimento dello snake è determinato esclusivamente dal "game clock": ogni volta che c'è un tick il serpente si muove di una casella indipendentemente dalla velocità con cui vengono schiacciati i tasti direzionali. Il tick dell'orologio di gioco cambia a ogni nuovo livello, permettendo di avere velocità dello snake sempre più maggiori man mano che il punteggio aumenta, tenendo però conto della crescente difficoltà dei livelli.

Possiamo dare una spiegazione di come le due macchine a stati finiti interagiscono tra loro, grazie alla memoria chiamata GAMEBOARD (la tabella di gioco), analizzando lo schema a blocchi dell'immagine.



Schema interazione tra FSM0 e FSM1

La macchina FSM1 accede alla gameboard sia in lettura che in scrittura: legge dalla memoria ciò che c'è scritto nel gioco (stato attuale), ovvero quali elementi (serpente, muro, frutto…) devono comparire nella mappa (si può dire che legge come il gioco si dovrà sviluppare) e modifica la gameboard (scrittura) in base a ciò che deve succedere (determinato dagli eventi). Quando ha terminato avverte la macchina FSM0 che, leggendo lo stato della gameboard, applica i cambiamenti disegnando sul frame buffer.

Riassumendo: la FSM0 legge il valore di ogni singolo quadratino costituente la mappa di gioco e disegna di conseguenza su frame buffer ciò che dovrà poi apparire sullo schermo, mentre la FSM1 gestisce il gioco vero e proprio decidendo e impostando ciò che apparirà a video.

Passiamo, a questo punto, alla descrizione più dettagliata delle singole macchine a stati.

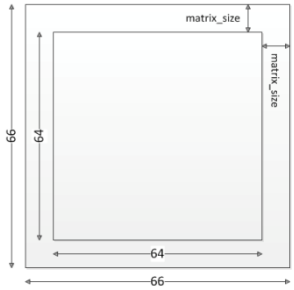
FSM0.

Iniziamo con il fornire alcune informazioni di carattere generale: si è scelto di utilizzare dei blocchi base (dimensione degli oggetti quali frutta, pezzo di serpente, etc) di dimensioni 1×1 celle elementari.

Lo snake ed i muri sono celle piene monocromatiche, a differenza della frutta che ha i bit_map da 5x5pixel. Nel codice, la dimensione del blocco base è identificata con matrix_size e forzandola a 1, ogni blocco può avere un solo contenuto nella sola cella elementare disponibile.

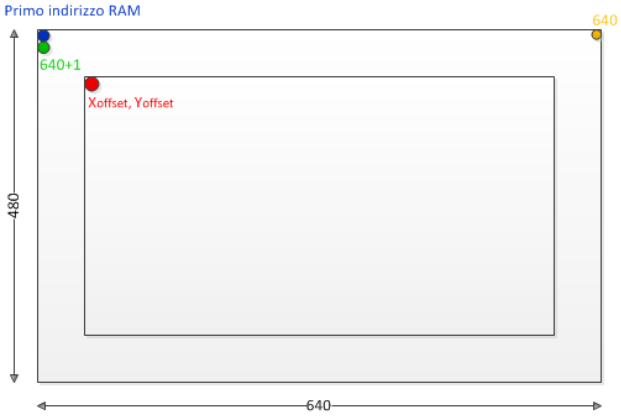
Non era obbligatorio fare una scelta di questo tipo, e si potevano anche utilizzare delle celle di dimensioni differenti, semivuote o più complesse, come ad esempio quelle 3×3 del Tetris.

Dimensioni quadro di gioco.



Per quanto riguarda le dimensioni del quadro di gioco, rimandiamo all'immagine. La dimensione 66×66 e l'introduzione di un "bordo", sono rimaste dal modulo del Tetris ed erano state introdotte per poter valutare alcuni casi particolari che si presentavano nella rotazione di alcuni pezzi se questa veniva effettuata vicino al bordo (dopo essere stati ruotati parte dei tetramini finiva fuori campo gioco).

Come già detto in precedenza la macchina a stati FM0 prende i dati sullo stato attuale del gioco, che sono le etichette (scritte e non numeri) di tempo e punti, il prossimo frutto e lo stato della gameboard (cosa contiene ogni quadratino) e scrive nel frame buffer di conseguenza. Per spiegare come lo fa facciamo un esempio: la gameboard inizia alle coordinate X_{offset}, Y_{offset}, quindi per potervi accedere, nel frame buffer, bisogna posizionarsi all'indirizzo $X_{offset} + Y_{offset} \times X_{resolution}$ con $X_{resolution} = 640$. La figura sottostante spiega questo particolare.



X_resolution = 640

Se ogni punto è un indirizzo separato si ha che:

Address (X_offset + Y_offset) = X_offset + Y_offset × X_resolution (che sono tutte le righe sopra X_offset).

Grazie a quanto detto sugli indirizzi posso sempre sapere in quale determinato punto della mappa sono posizionato.

Arrivati a questo punto del progetto è insorto un problema di allineamento delle immagini e, in particolare dei muri che delimitano lo spazio di gioco, dovuto alla scelta di impostare nella gameboard, le caselle che compongono il campo, in griglie da 5×5 pixel (block_size) spaziate di 1 pixel (block_gap).

Nel dettaglio: la memoria RAM utilizzata è a indirizzamento lineare, ed è quindi composta da indirizzi compresi tra 0 e 2^n, con n il numero di bit al loro interno. Prima di poter, quindi, leggere o scrivere dei dati nella memoria, è necessario specificare l'indirizzo su cui si vogliono effettuare le operazioni.

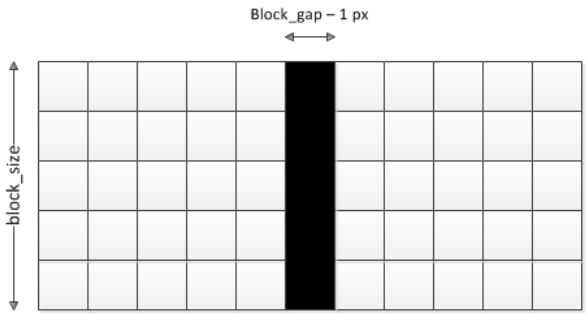
Nel nostro caso abbiamo 640×480 pixel totali e 16 colori possibili mentre la memoria della scheda è fornita di bus dati a 16 bit. Per ogni indirizzo fisico della memoria si possono quindi leggere 16 bit che corrispondono a 4 pixel. In altre parole questo vuole dire che ogni volta che mi sposto di un indirizzo ho accesso a 4 pixel.

Partendo ora dalle coordinate X_{offset}, Y_{offset} mi domando cosa c'è scritto nella gameboard all'indirizzo corrispondente: prendiamo il caso in cui non ci sia scritto nulla ovvero la casella è vuota. Nella RAM dovrà esserci quindi il valore "nero", associato al background del campo di gioco. Si devono quindi mettere 5×5 pixel neri in quanto, tramite la variabile "block_size", si è impostata ogni casella (di cui è composta la schermata di gioco) di tali dimensioni.

Per fare ciò nella RAM si scriverà fino a quando non saranno presenti 5 pixel neri, che, in pratica, significa incrementare di volta in volta l'indirizzo della memoria e scrivere "nero – nero – nero – nero - nero".

Il procedimento descritto è in realtà più complesso infatti se ogni indirizzo ha 4 pixel per scrivere 5 volte il valore "nero" bisognerà andare all'indirizzo di coordinate X_{offset}, Y_{offset}, porre a "nero" tutti i 16 bit disponibili (4 pixel), spostarsi all'indirizzo successivo e in questo scrivere solo i primi 4 bit (1 pixel) neri, per un totale di 5 pixel. Si è colorata così una casella.

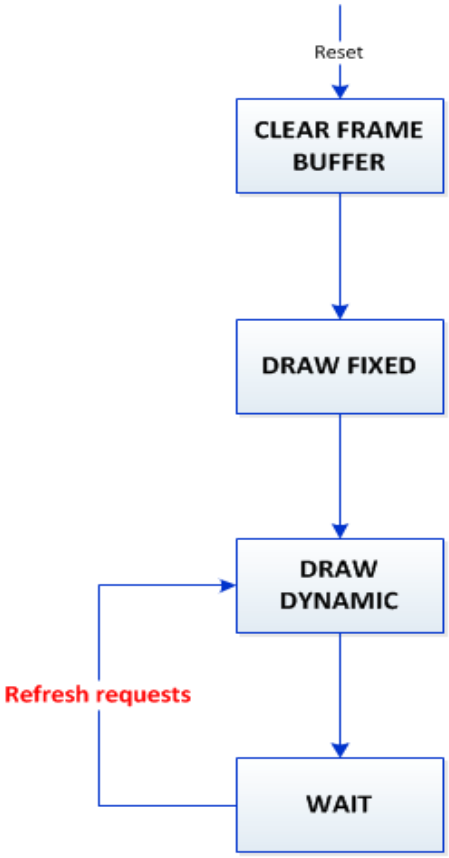
Si prosegue valutando cosa c'è nella casella successiva della gameboard: è presente un muro. All'indirizzo in cui ci si trova, si avranno quindi 4 bit neri, a causa della colorazione della precedente casella, a cui verranno fatti seguire 3 bit bianchi, il colore dei muri. Infine, analogamente a quanto spiegato in precedenza, ci si sposterà di un indirizzo in cui imposteremo i primi 2 pixel bianchi. Si procede fino alla fine della riga per poi tornare indietro e iniziare quella seguente.



Il problema è diventato ancora più complesso con l'inserimento tra le varie caselle di un gap, visualizzato dal giocatore come un piccolo spazio nero, di un pixel (Figura a lato).

Grazie a quanto descritto si sono risolti il problema di disallineamento, anche sé probabilmente scegliendo celle da 4×4 (o multipli di 4) pixel e non inserendo il gap, la problematica analizzata sarebbe stata meno complessa se non addirittura inesistente.

A completamento del discorso sulla macchina FSM0, diamo una spiegazione schematica del suo funzionamento.



- **CLEAR FRAME BUFFER:** Cancella tutta la SRAM.
- **DRAW FIXED:** Disegna tutto ciò che rimane fisso fin dal primo frame, come i bordi del campo da gioco e le scritte, e che una volta inserito nella RAM non viene più toccato.
- **DRAW DYNAMIC:** Disegna tutti gli elementi dinamici come le cifre di tempo e punti, i muri dei labirinti e il disegno del prossimo frutto che comparirà sullo schermo.
- **WAIT:** attende richiesta di refresh esterna.

REFRESH REQUEST: viene triggerata e può essere di due tipi:

1. TIPO A: triggerata da un clock a 1 Hz, ovvero ogni secondo viene comunque aggiornata tutta la grafica. Serve principalmente al display del tempo di gioco (elapsed time) per fare in modo che la grafica non si aggiorni comunque ogni secondo.
2. TIPO B: triggerata dalla seconda macchina, FSM1, con una richiesta di UPDATE. Quando è stato deciso il nuovo stato della mappa di gioco e sono stati posizionati tutti gli elementi al suo interno, la FSM1 richiede un update alla FSM0, che lo esegue.

FSM1.

Per una migliore comprensione della macchina a stati FSM1, iniziamo col dare un'idea su come i dati vengano registrati all'interno della gameboard, la tabella di gioco memorizzata in RAM.
L'implementazione è con indirizzamento lineare, di 66x66 locazioni, con larghezza di parola 15 bit e, vista la quantità non indifferente di dati, si è preferito usare una RAM piuttosto che dei registri; per risolvere il problema dell'accesso condiviso fra le due FSM è stata usata una memoria di tipo dual-port.
Si hanno, quindi, 15 bit totali per ogni cella: i **primi 2 bit**, quelli meno significativi, indicano che cosa c'è all'interno della casella considerata. In particolare:

- 00 = casella vuota.
- 01 = serpente.
- 10 = frutta.
- 11 = ostacolo (muro).

La mappa di ogni livello, cioè la gameboard al caricamento di un nuovo livello, sarà composta da gruppi di 15 bit, con i primi 2 a valore 00 o 11, per indicare rispettivamente la presenza di un muro o di una casella vuota, e tutti i rimanenti bit, da 2 a 14, posti a 0.

Inoltre, nel caso in cui, mi trovassi all'indirizzo associato alle coordinate def_X, def_Y, che sono le coordinate di default in cui si è scelto di posizionare il serpente, i primi 2 bit verranno precaricati al valore 01. Ciò da modo al giocatore, prima dell'inizio di una nuova partita o di un nuovo livello, di vedere la posizione di partenza dello snake. Senza questo accorgimento, infatti, non sarebbe possibile prevedere in quale parte del campo comparirà il serpente.

Si sono anche inseriti 2 secondi di delay prima dell'inizio del gioco, per dare tempo al giocatore di rendersi conto dell'architettura del livello caricato (blocco "Reset_delay").

Come già detto, i **bit da 2 a 14** sono inizialmente posti tutti a 0. Nel caso in cui però, l'oggetto contenuto nella casella sia un frutto, primi 2 bit = 10, i bit successivi danno importanti informazioni. In particolare:

- bit da 2 a 4 contengono un indice (**INDEX**)
- bit da 5 a 7 danno informazione sul tipo di frutto.

L'index mi permette di ottenere tutte le informazioni riguardanti il frutto che è appena stato mangiato. Tutta la frutta è infatti registrata in un array chiamato "fruit_array" di 8 posizioni indirizzabile con 3 bit. Si hanno 8 posizioni in quanto sullo schermo si possono avere al massimo 8 frutti contemporaneamente, e non di più.

Ogni cella del "fruit_array" contiene quindi informazioni sul frutto in essa salvato, ovvero, in ognuna delle 8 posizioni viene scritto il tipo di frutta associato ai vari insdici. Per ciascun indice si possono conoscere i punti da assegnare (ogni frutto da al giocatore un numero diverso di punti), l'aumento della lunghezza dello snake associato al tipo di frutto, da quanto tempo il frutto è in campo (anche questo parametro varia in base al tipo), quanto tempo deve ancora rimanerci, ecc…

In pratica, ogni volta che lo snake mangia un frutto, viene chiesto quale, tra tutti i frutti presenti sullo schermo, è stato mangiato e, la risposta è: << quello contenuto nella posizione INDEX del "fruit_array">>.

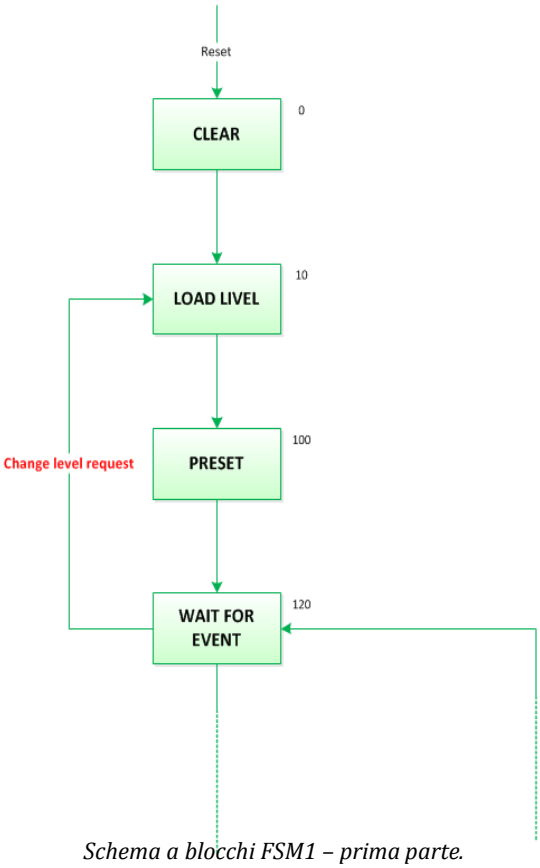
Notare che, se,ad esempio, sullo schermo sono presenti solo banane, l'index mi dice, non solo che il tipo di frutto è "banana", ma anche quale delle banane visualizzate è stata mangiata.

Rappresentazione frutta in gameboard.

Per quanto riguarda invece i bit da 5 a 7 essi contengono volutamente un'informazione ridondante, è possibile infatti risalire al tipo di frutto anche grazie all'index di cui sopra. Si sarebbe potuto ottimizzare questo aspetto del progetto, ma si è deciso di mantenere la ridondanza (anche vista la presenza di bit inutilizzati) per facilitare il

lavoro alla macchina a stati FSM0, che, dopo aver letto dalla gameboard e aver deciso cosa scrivere nel frame buffer, non deve recuperare l'informazione sul tipo di frutto passando attraverso l'index ma trova già l'informazione pronta, risparmiando così tempo.

Nel caso il tipo della cella sia "snake" (primi 2 bit = 01), i **bit da 2 a 14** contengono il puntatore (indirizzo nella gameboard) al pezzo successivo del serpente come verrà spiegato successivamente.



Schema a blocchi FSM1 – prima parte.

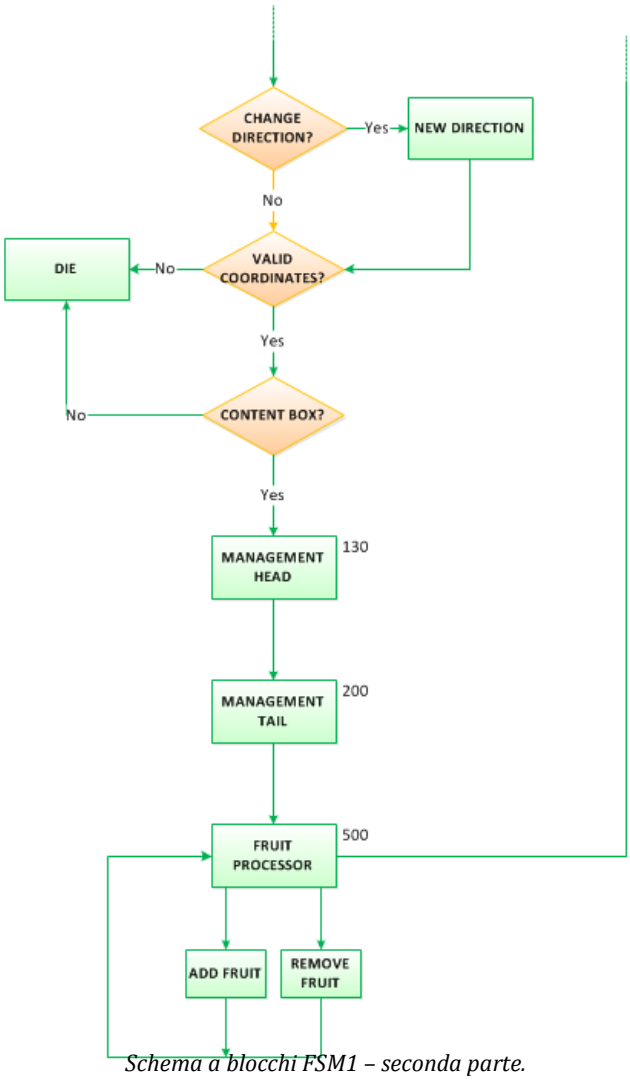
Vediamo nel dettaglio il funzionamento della macchina a stati FSM1 aiutandoci con uno schema a blocchi, che analizzeremo diviso in due parti.

Prima parte:

- **CLEAR:** fa una pulizia di tutto ciò che c'è (registri, gameboard), anche dei punti.
- **LOAD LIVEL:** carica il livello dalle mappe presenti nella ROM, per cui avremo che il livello 0 inizia all'indirizzo 0, il livello 1 all'indirizzo 64×64, e così via. Ogni livello inizia quindi 4096 (64×64) indirizzi dopo rispetto al precedente, eccezione fatta per il primo che parte da 0. Viene preservato il punteggio.
- **PRESET:** una volta caricato il nuovo livello, viene presettato lo stato del gioco e quindi vengono impostate le coordinate di default dello snake, la direzione di inizio (nel nostro caso è dx) e la lunghezza (= 1). Viene inoltre inserito valore nullo in corrispondenza della variabile che indica l'allungamento del serpente e viene eliminata tutta la frutta; viene eliminata qualsiasi operazione pendente (aggiunta/rimozione frutta, cambi direzione, etc), ed effettuata infine una richiesta di aggiornamento grafico alla FSM0.
- **WAIT FOR EVENT:** l'evento che la macchina a stati aspetta può essere un "tick" dell'orologio di gioco, in seguito al quale passa allo stato successivo riportato nella seconda parte dello schema a blocchi, oppure una richiesta di cambio livello (**CHANGE LEVEL REQUEST**) che riporta allo stato 10.

Seconda parte:

- **CHANGE DIRECTION? :** viene verificata la presenza di una richiesta per un cambio di direzione del serpente.



Schema a blocchi FSM1 – seconda parte.

- **NEW DIRECTION:** se è arrivata una richiesta di cambio di direzione ovvero è una latch request, viene inviato un messaggio di conferma e aggiornato il registro direzione con il nuovo verso che il serpente dovrà seguire.
- **VALID COORDINATES:** sia nel caso in cui lo snake cambi direzione oppure mantenga la stessa (non viene premuto alcun tasto), viene effettuato un controllo sulla validità delle future coordinate in cui dovrà spostarsi la testa (devono trovarsi all'interno dell'area di gioco).
- **CONTENT BOX:** se le coordinate sono invece valide ed è quindi possibile effettuare lo spostamento, è necessario controllare il contenuto della casella: è un frutto, è vuota oppure contiene qualcosa di solido come un muro o lo snake? Nell'ultimo caso la partita termina in quanto il serpente ha sbattuto contro se stesso o un muro, altrimenti viene effettuato lo spostamento grazie ai blocchi successivi di gestione della testa e della coda.
- **DIE:** la partita termina immediatamente con la morte.
- **MANAGEMENT HEAD:** si occupa di spostare la testa del serpente nella casella successiva. Lo snake è gestito come una lista memorizzata nella gameboard. Ciascun elemento della "list" è accessibile specificandone l'indice che corrisponde al suo indirizzo nella gameboard ("list[index]")

  Il dato contenuto in RAM (gameboard) contiene il tipo dell'elemento indirizzato (i primi due bit, list[index].type, fissi a 01 per il serpente) e il puntatore al prossimo elemento (cioè l'indirizzo del prossimo pezzo di snake, list[index].nextvalue). Per la gestione della struttura dati vengono inoltre utilizzati due ulteriori variabili: "startindex" per definire l'indice/indirizzo di testa della lista e "endindex" per quello di coda (in due registri a 13 bit).

Ricordiamo che i blocchetti componenti lo snake sono aggiunti durante il gioco e ,come anche quelli di tipo "wall" raffiguranti i muri, sono "hard" e portano quindi alla morte.

Nell'avanzamento dello snake, l'indirizzo nel quale si trova la testa non punta a nulla. Se le nuove coordinate risultano valide, si imposta il tipo del blocchetto indirizzato da esse in "snake"; successivamente si imposta il nuovo indirizzo come indice di testa mentre il precedente elemento di testa punta al nuovo elemento. La testa è ora nella nuova posizione e si può incrementare la variabile con la lunghezza del serpente.

CODICE DI GESTIONE TESTA.

```
funzione addhead(newindex):
list[startindex].nextvalue=newindex
list[newindex].type=snake
length++
startindex=newindex
```

Nella gameboard si avrà quindi che i bit, all'indirizzo in cui si trova la testa prima di essere spostata, saranno organizzati come da figura, dove NEXT POINTER rappresenta le nuove coordinate:

- **MANAGEMENT TAIL:** include la gestione dell'allungamento della coda. Un contatore (impostato inizialmente quando viene mangiato un frutto) indica di quanti blocchi deve ancora allungarsi il serpente. Se il serpente non deve essere allungato (contatore = 0) viene semplicemente tolta la coda decrementando di un'unità la variabile contenente la lunghezza dello snake. L'indice di fine snake quindi, diventa quello che era puntato dal precedente ultimo elemento ormai eliminato.

  CODICE DI RIMOZIONE CODA.

  ```
  removetail
  length--
  endindex=list[endindex].nextvalue
  list[endindex].value=empty
  ```

  Se invece il serpente deve essere allungato, non viene tolta la coda ma viene decrementato il contatore con gli "allungamenti residui".
- **FRUIT PROCESSOR:** Gestisce la frutta sullo schermo. Se nell'ultima mossa è stato mangiato un frutto devo rimuoverlo e per sapere quale tra quelli visualizzati utilizzo l'index già introdotto precedentemente. Se invece il frutto va eliminato perché è scaduto il tempo (timer esterno) in cui poteva rimanere sul campo di gioco, verrà utilizzato il parametro auto_remove impostandolo al valore dell'index del frutto da eliminare. Analogamente un timer esterno può inviare una richiesta di aggiungere un frutto.

  I processi di rimozione e di aggiunta della frutta vengono spiegati nei due blocchi seguenti.

  Tutte le operazioni sulla frutta sono effettuate dopo aver portato a termine tutto il resto.
- **ADD FRUIT:** prendiamo il caso in cui arrivi una richiesta di aggiungere un frutto ma sullo schermo ci sai già il numero massimo di frutti possibili (8). Si fa credere alla macchina di aver comunque aggiunto un frutto sullo schermo anche se ciò non è avvenuto, altrimenti aspetterebbe

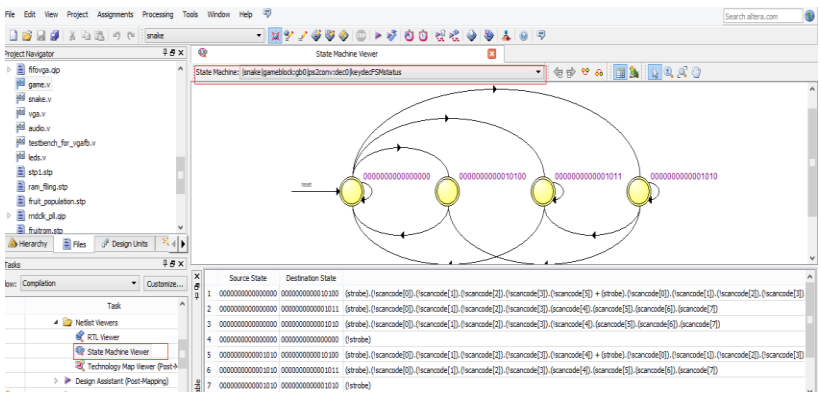
all'infinito, e si ritorna allo stato in cui viene controllato se ci sono altre richieste di aggiunta o rimozione. Il controllo è importante per poter servire tutte le richieste in quanto possono essercene più di una (es: frutto mangiato assieme a due rimozioni automatiche e un'aggiunta) ma è possibile servirne solo una alla volta.

Se invece abbiamo a video meno di 8 frutti allora possiamo aggiungerne uno. Per farlo ci serviamo di un generatore di numeri casuali per generare la posizione del frutto da aggiungere. Una volta generate le coordinate è necessario assicurarsi, con un controllo nella gamenboard, che queste siano valide ovvero che non corrispondano ad una casella piena. In questo caso posso inserire il frutto. Al contrario se le coordinate scelte non sono valide vengono generati altri numeri random.

Una volta aggiunto il frutto nuovo (il tipo era previsualizzato in basso a destra sullo schermo), si genera con un altro generatore casuale il tipo del prossimo frutto che verrà aggiunto.

- **REMOVE FRUIT:** le situazioni in cui va rimosso un frutto sono due: il timer è scaduto o il frutto è stato mangiato. Nel primo caso il timer fornisce anche l'indice del fruit_array del frutto da rimuovere (auto_remove): ciò permette di estrarre dall'array l'indirizzo del frutto nella mappa di gioco. Si procede quindi a cancellarle il contenuto della gameboard al suddetto indirizzo, ovvero si mette tutto a colore di background (nero). Per concludere si elimina tutto ciò che è relativo al frutto nell'array (presenza, timers, etc). Questo è possibile grazie alla reciprocità tra la gameboard e l'array: nella memoria viene salvato l'indice dell'array e in quest'ultimo viene a sua volta scritta la posizione nella gameboard. Nel caso di frutto mangiato invece si salta il procedimento di pulizia dalla gameboard in quanto al posto della frutta si è già inserita la testa del serpente (nello stato "management head") e si procede alla sola rimozione dall'array. Ricordiamo infine che la frutta compare in momenti casuali e in maniera più frequente se non è presente nessun frutto in campo.

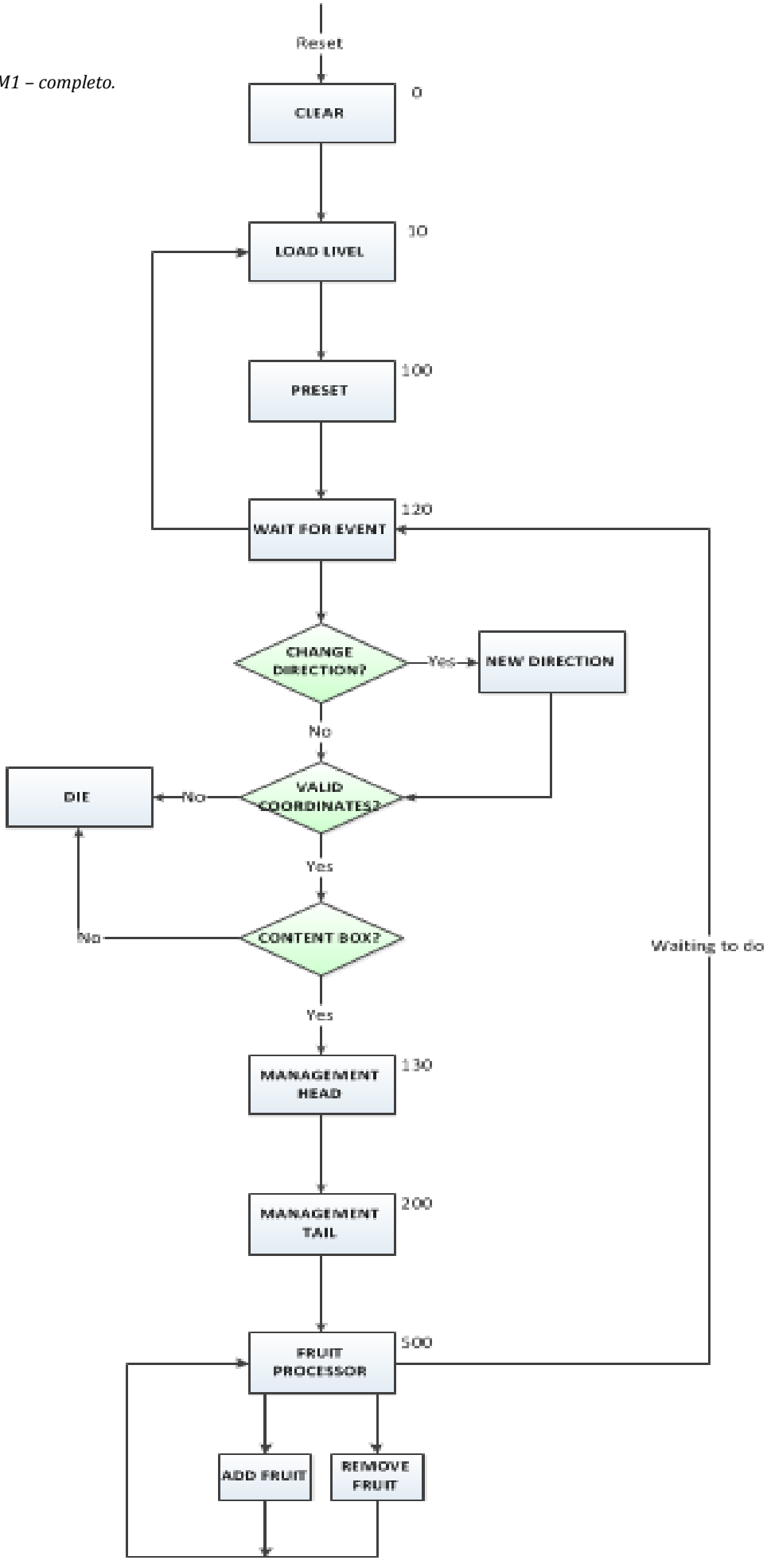
Riportiamo di seguito, un'immagine della visualizzazione di una macchina a stati finiti nel tool di sviluppo Quartus e, a pagina seguente, lo schema a blocchi completo della macchina FSM1.



Macchina a stati finiti visualizzata nel tool Quartus.

Non è stato possibile riportare la visualizzazione delle macchine descritte in quanto essendo di dimensioni elevate e presentando quindi molti stati il tool di sviluppo non è in grado di aprirle.

Schema a blocchi FSM1 – completo.



Per lo sviluppo dell'intero progetto è stato necessario l'utilizzo di molte altre macchine a stati e semplici moduli, per poter allargare le funzionalità del gioco. Tra tutti si annoverano:

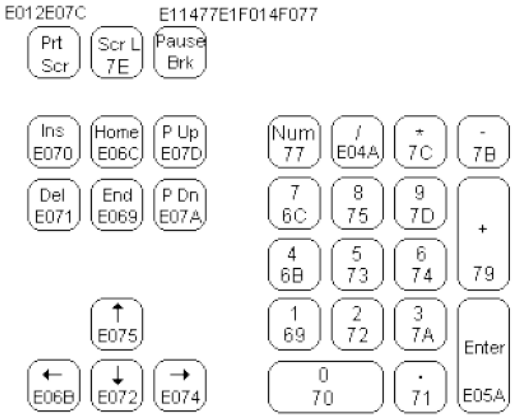
- macchina a stati finiti per la codifica dei comandi provenienti dalla tastiera e dai push buttons.
- module ROMTOPALETTE per la conversione del formato dati RAW nel formato compatibile scelto, ovvero quello a 16 colori;
- module BCDEXPANDER per visualizzare sue due display a 7 segmenti: uno rappresenta il livello corrente, l'altro il numero di frutti presenti attualmente.
- module RNDGEN che si occupa della generazione di un numero casuale per la generazione della frutta, con qualche accorgimento per popolazione della stessa come ad esempio limitare l'eccessiva presenza di un unico frutto (in più versioni a seconda dei bit da generare).
- module GTG per la generazione del clock degli eventi

PS2_CONTROLLER.

Il modulo gestisce la ricezione dei comandi dalla tastiera.

Di base l'idea è quella di associare dei particolari segnali, gli scan-codes, ad un determinato movimento del serpente.

Scan-codes estesi dei tasti con le frecce usati nel progetto



Lo scan codes non è altro che uno standard di riconoscimento del tasto premuto (possiamo considerarlo un parallelismo all'ASCII). I tasti estesi richiedono due scan codes differenti: E0 e NN. Quando si rilascia un tasto, viene inviata la sequenza F0-NN e, per un esteso, E0-F0-NN.

Il blocco di base del processore comandi tastiera è stato ripreso dal Tetris ma si sono apportate significative modifiche al fine di renderlo più efficiente e adatto al nostro gioco.

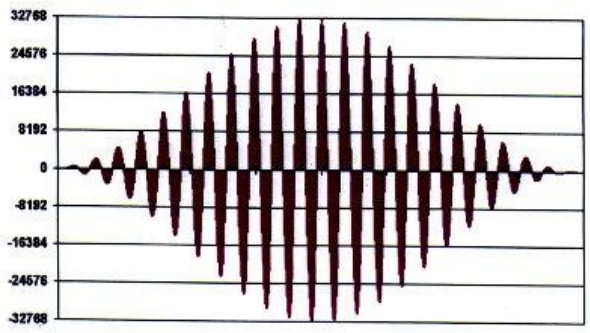
In figura, gli IP Cores (Intelectual Property) scaricati dal sito di Altera University Program che permettono di interfacciare la tastiera PS/2 con la board e di gestire il protocollo.

Prima di tutto si è ottimizzata la macchina a stati che analizza lo scan code, modificandola in modo tale che vengano memorizzati solamente i dati provenienti dalla pressione del tasto e non quelli dovuti al rilascio. Questo ha risolto molti problemi di sensibilità dei tasti presenti nel Tetris, permettendo di ottenere maggiore fluidità e precisione nel controllo dello snake.

È stata inoltre apportata un'ulteriore modifica con l'inserimento di un **BUFFER A 2 COMANDI** per la gestione di comandi ripetuti, qualora fossero necessari spostamenti rapidi del serpente.

In Tetris per ogni comando da tastiera che arrivava veniva subito triggerato un evento, ovvero se ad esempio veniva premuto più volte il tasto destra, il tetramino si spostava velocemente in questa direzione senza aspettare il successivo tick dell' orologio di gioco (5 Hz).
Nel nostro caso invece ciò non avviene: lo snake si muove sempre alla velocità del game_clock, indipendentemente dalla velocità con cui vengono schiacciati i tasti direzionali.
Prendiamo adesso il caso in cui lo snake stia andando verso sinistra e il giocatore voglia fare un movimento veloce nelle direzioni su-destra. Poiché, come sappiamo, siamo vincolati dall'orologio di gioco verrà prima eseguita la mossa "su" e, al tick successivo, la mossa "destra". Per poter fare ciò si è aggiunto appunto un buffer che permette di salvare sempre in memoria 2 comandi che vengono poi eseguiti uno di seguito all'altro man mano che il serpente si muove. In altre parole si è aggiunto un array temporaneo nel quale vengono memorizzati gli scan-codes associati ai tasti premuti, per poi essere "rilasciati" all'inizio del prossimo colpo di clock di gioco (game tick). Ad ogni colpo di clock viene eseguito un evento singolo, e vengono mantenuti nel buffer al più due eventi.
Affinchè funzionasse tutto correttamente si sono inoltre dovuti considerare alcuni casi particolari come, ad esempio, se la direzione attuale è sinistra e il giocatore preme destra-su, la prima mossa, che non è possibile effettuare (per cambiare direzione da sx a dx il serpente deve necessariamente andare prima verso l'alto), deve venire ignorata. Prima dell'attuazione di ogni comando, si è quindi inserito un controllo sia sulla compatibilità tra direzione dello snake e prossima mossa, sia sulla compatibilità tra le due mosse salvate nel buffer.
Quando il buffer è completamente piene viene gettato via tutto. Ciò significa che se il giocatore schiaccia molti tasti velocemente o contemporaneamente può accadere che alcuni comandi, soprattutto gli ultimi, vengano persi e non eseguiti. In altre parole se il buffer è pieno i comandi successivi vengono ignorati e poiché il nostro buffer è stato creato per contenere fino a 2 mosse, dal dalla terza mossa in poi non viene considerato più nulla.

AUDIO_FSM:AUDIO.



Il file audio contenuto all'interno della memoria flash è formato da campioni PCM lineari a 16 bit, campionati a 32KHz. Questo tipo di formato permette di manipolare il segnale analogico con 2^{16}=65536 livelli, ovvero c'è una suddivisione in 65536 livelli del segnale (come da figura).
Il file audio è stato campionato a 32KHz, a

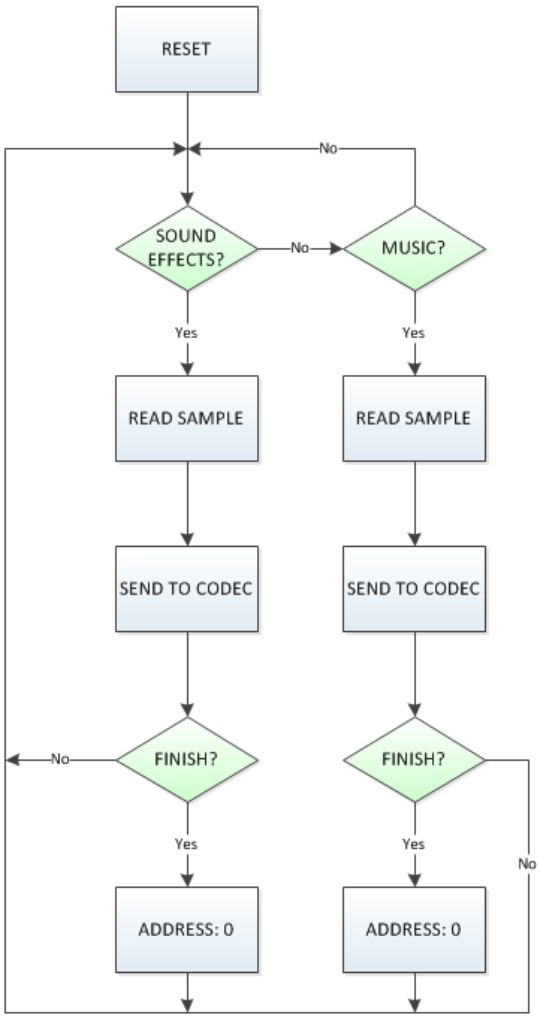
discapito della qualità, per ridurne la dimensione in Kbyte visto che la memoria flash può contenere solo fino a 4MB=4194304byte. Tutti i suoni, dalla musica di sottofondo ai vari effetti sonori, sono concatenati in un unico file.

STRUTTURA AUDIO:
32000kHz/kSa, 16bit, mono
MSB first (motorola)

SAMPLES:
x: OST 1711060 esimo byte
0: mangia 33024 esimo byte
1: levelup 35432 esimo byte
2: morte 262265 esimo byte

Una nota importante va fatta sul formato del file da introdurre nella flash. I più noti formati audio come mp3 o wav contengono all'interno (visualizzabile con qualsiasi HEX editor) un header che fornisce informazioni globali sul file, ma che non da alcuna rappresentazione in forma fisica dello stesso. Per eliminare questo header, e rendere il file utilizzabile nel progetto è stato necessario convertirlo in formato pcm (campioni grezzi, come un wav strippato dall'header), unico manipolabile tramite I2C tra le periferiche fisiche della board DE1.

Il pcm provvede a salvare i dati audio senza nessun tipo di compressione e la forma d'onda viene memorizzata direttamente così com'è, sia pure digitalizzata. Il file risultante (output.pcm) misura 4.083.562 byte e le operazioni su di esso sono state effettuate tramite software esterno Audacity.



Schema a blocchi FSM audio.

Il modulo "audio_fsm:audio", riceve in input le richieste provenienti dal blocco gameblock e restituisce come output i segnali audio in funzione dell'evento gestito. Indirizza inoltre la memoria flash per prelevare i campioni che poi verranno mandati al codec grazie alla FIFO.

Quando il gioco è attivo (non in pausa, non morte) la musica viene riprodotta ciclicamente. Al sopraggiungere di un evento la musica si interrompe e si passa alla riproduzione dell'effetto sonoro relativo all'evento accaduto: mangia, cambio livello o morte.

Diamo quindi una descrizione della macchina a stati del modulo.

La memoria flash è fornita di bus dati da 8 bit ma, visto che ognuno dei nostri campioni pesa 16 bit, è possibile leggere solo ½ campione alla volta ed è necessario quindi fare due letture della flash per poter comporre un intero campione (aspettando i necessari tempi, come indicato nel datasheet della flash). Il

procedimento con cui viene fatta tale ricostruzione è il seguente: inizialmente la macchina a stati finiti non scrive nulla nella FIFO, il cui indirizzo iniziale è 0, e aspetta.

Se la FIFO non è piena la FSM prosegue prendendo i dati dal primo indirizzo della memoria flash, quelli del byte più significativo (l'audio data è inserito a sinistra), viene poi incrementata la flash di un'unità e si inserisce in coda ai dati precedentemente prelevati quelli a cui si ha appena avuto accesso.

Vengono quindi presi i primi 8 bit e incodati i secondi 8. È stato composto un intero campione (16 bit).

Dopo averlo ricostruito si passa alla scrittura del campione all'interno della FIFO: se quest'ultima non è piena si scrive una seconda volta. Viene incrementato il contatore dei campioni riprodotti ("playati") e se questo risulta essere uguale al numero di campioni totali ma è ancora presente l'abilitazione della musica allora la memoria flash ritorna al valore 1 e il sonoro continua. Altrimenti, nel caso in cui la musica venga interrotta, la flash ritorna a 0 e aspetta un'eventuale richiesta di effetto sonoro oppure fa riprendere la musica di sottofondo. Se serve l'effetto sonoro viene controllata la richiesta, si carica il suono corrispondente all'effetto desiderato e viene infine dato l'ack.

Finita la riproduzione (dell'effetto) viene valutato se c'è bisogno di un ulteriore effetto o se invece deve riprendere la musica. Nell'ultimo caso l'audio di sottofondo del gioco riprende esattamente da dove si era fermato prima dell'effetto sonoro, senza ricominciare da capo.

Infine se durante la riproduzione di un effetto arriva una richiesta per un altro di questi, il primo viene bloccato immediatamente per poter attivare il secondo: se per esempio il giocatore ha appena mangiato un frutto e subito dopo cambia il livello l'effetto associato all'evento "mangia" si interrompe e parte la riproduzione del suono di cambio livello.

PALETTE.

Questo modulo definisce la tavolozza (mappatura dei colori eventualmente modificabile, detta anche ''palette'') dei 16 colori a 4 Bit scelti nell'insieme di quelli disponibili. Il modulo prevede in ingresso un bus di 4 bit che vengono poi trasformati in un bus 12 bit RGB, come mostrato in tabella.

0 — black	8 — gray
1 — maroon	9 — red
2 — green	10 — lime
3 — olive	11 — yellow
4 — navy	12 — blue
5 — purple	13 — fuchsia
6 — teal	14 — aqua
7 — silver	15 — white

TAVOLOZZA COLORI A 4 BIT.

- *0 = nero*
- *1 = marrone*
- *2 = verde*
- *3 = verde oliva*
- *4 = blu scuro*
- *5 = viola*
- *6 = foglia di the*
- *7 = argento*
- *8 = grigio*
- *9 = rosso*
- *10 = lime*
- *11 = giallo*
- *12 = blu*
- *13 = fuchsia*
- *14 = turchese*
- *15 = bianco*

KITT.

Questo blocco di codice è stato inserito per rendere l'effetto ''supercar'' (scorrimento avanti e indietro) dei 10 led rossi disponibili sulla board.
L'effetto si è ottenuto traslando a destra e a sinistra un registro (ledbus) i cui bit corrispondono allo stato dei LED: spostando un bit avanti e indietro si fa scorrere il LED acceso proprio avanti e indietro.

CONCLUSIONI.

Il gioco è stato realizzato totalmente in linguaggio Verilog e la gestione sequenziale gestita interamente da macchine a stati finiti senza l'ausilio di alcun processore.
Lo sviluppo del progetto ha messo in evidenza come sia possibile realizzare un videogioco, anche abbastanza complesso come "Snake", sfruttando una programmazione esclusivamente di tipo hardware. Ci ha dato modo non solo di avvicinarci ma, sopratutto, di approfondire in dettaglio, il funzionamento di varie periferiche come schermi, grazie allaVGA, tastiere PS/2, dispositivi audio, e dei protocolli per gestirli. Abbiamo inoltre aumentato la nostra conoscenza in merito ad argomenti tecnici quali i vari tipi di memorie (SRAM, flash, FIFO) e i meccanismi di comunicazione tra di esse, nonché al funzionamento e alla progettazione di macchine a stati finiti.
Le **PROBLEMATICHE** che sono insorse durante tutto lo sviluppo del gioco ci hanno permesso non solo di acquisire una discreta conoscenza del linguaggio Verilog HDL, ma anche di migliorare la nostra capacità di analisi e risoluzione di problemi più o meno complessi. Tra questi i più significativi, che hanno richiesto il maggiore impegno nella risoluzione, sono sicuramente l'adattamento delle caselle di 5×5 pixel ad una memoria con indirizzi da 4 pixel (è stato necessario riscrivere molte righe di codice), l'introduzione del buffer a due comandi e l'inserimento della parte audio.
Il gioco è piuttosto spartano ma è un ottimo esempio di cosa è possibile fare con tecnologie come la scheda DE1 e l'FPGA. Gli elementi che ci hanno danno maggiore soddisfazione sono: possibilità di visualizzare un campo da gioco su monitor VGA standard e di poter giocare con pushbutton o tastiera PS2 a piacimento (grazie ad uno switch), opzione di mettere in pausa e riprendere il gioco da dove si è interrotto e presenza sia musica di sottofondo che di effetti sonori (con possibilità di disabilitarli). Tutta la configurazione è immediata grazie all'utilizzo di switches.
Seppure la piattaforma di sviluppo sia una DE1 di ALTERA è possibile portare il gioco su altre architetture FPGA a patto di sostituire i componenti "megafunction" specifici Atera (PLL e memorie) con equivalenti.

Allo stesso modo è possibile pensare di realizzare un ASIC senza troppe difficoltà.

Il progetto è nato come sviluppo del Tetris e in particolare della parte audio ma presenta notevoli differenze e modifiche rispetto a questo. Tra le più importanti:

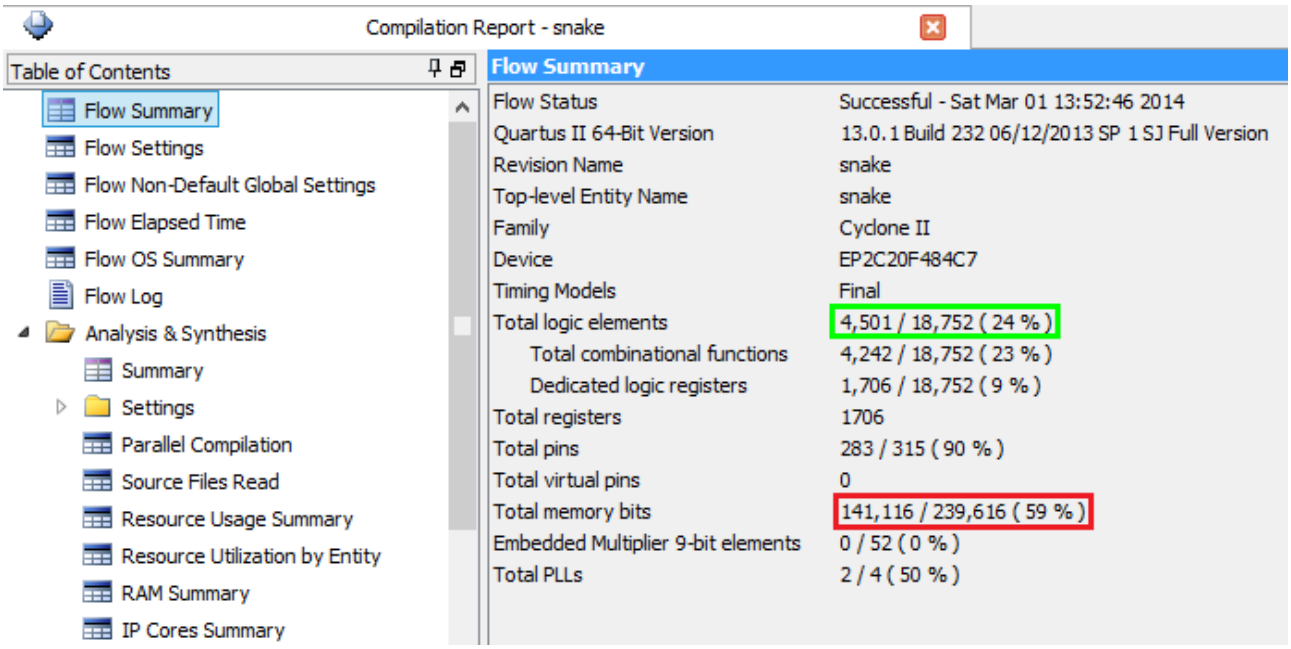
- VGA: aggiunto uno switch di calibrazione per la configurazione corretta del monitor che risultava non ben posizionato sullo schermo (generazione di un frame bianco)
- AUDIO: aggiunta del sistema di riproduzione della musica di sottofondo integrato a quello di riproduzione dei vari effetti sonori
- BUFFER a due comandi per i movimenti veloci dello snake.

L'approccio modulare utilizzato, in cui tutti i moduli sono autonomi e parlano separatamente tramite req ed ack, permette di utilizzarli anche separatamente, mentre l'inserimento dei file resources in memorie esterne permette flessibilità nell'eventualità di un cambio dell'hardware.

Abbiamo optato per la scelta dell'approccio modulare pensando all'ipotesi di sviluppo del progetto con la costruzione di un chip, in cui due fattori sono importanti:

1. la RIUTILIZZABILITÀ DEL CODICE e quindi del chip.
2. POSSIBILITÀ DI MODIFICA DELLE SINGOLE PARTI all'interno del progetto che permette ad esempio di utilizzare rom esterne modificando solo la relativa parte di codice piuttosto che riscrivere tutto il codice del chip. In termini di costo c'è un grosso risparmio anche a livello di componentistica.

Di seguito si riporta un riepilogo delle risorse utilizzate dalla board DE1 mediante la compilazione e la simulazione del progetto.



Si può notare il ridotto utilizzo di elementi logici (Logic Elements, LEs) messi a disposizione dalla board DE1, che non supera il 25% (nella simulazione raggiunge il 24%).
La voce "Total memory bits" che rappresenta il consumo di memoria RAM (o ROM se il progetto viene caricato in modalità Active Serial Programming) nella DE1, si posiziona su livelli intermedi, ma comunque non classificabili come basso consumo. A tal proposito si potrebbe pensare di far uso di una memoria RAM (o ROM) esterna, tenendo in considerazione anche quanto detto in precedenza riguardo la possibile implementazione di un ASIC, in modo da rilasciare alcune risorse dell' FPGA.

SVILUPPI FUTURI.

A conclusione del lavoro svolto diamo qualche spunto per possibili sviluppi futuri del gioco:

- **A.** aggiunta di nuovi frutti, con possibilità di scegliere a piacere la grandezza e la forma;
- **B.** aggiunta delle scritte di Game Over, Pause e Level Number per arricchire la grafica del gioco;
- **C.** interfacciare il NES Controller da usare al posto della tastiera per poter giocare;
- **D.** portare il progetto sulla scheda Altera DE2, con la possibilità di aumentare la risoluzione della grafica e usare la memoria SDRAM invece della SRAM per un framebuffer più grande (le modifiche da fare in questo caso sarebbero poche);
- **E.** utilizzo della metà libera della RAM, ottenuta grazie al fatto che non abbiamo utilizzato tecniche di double buffering, per apportare alcune migliorie come l'aumento della risoluzione;
- **F.** fare in modo che la testa e la coda del serpente non risultino quadrattini ma abbiano forma differente rispetto al resto del corpo;
- **G.** inserimento di musiche differenti per ogni livello.

BIBLIOGRAFIA.

- Slides del corso Elettronica II – FPGA del Prof. Marsi
- Cyclone II FPGA Starter Development Kit User Guide
- Cyclone II FPGA Starter Development Board Reference Manual
- IS61LV25616 High Speed Asynchronous CMOS Static RAM Datasheet
- http://www.infotart.com/blog/wp-content/uploads/2008/06/windows_4bit_color_swatches.png (Palette dei colori)
- http://retired.beyondlogic.org/keyboard/keybrd.htm (PS/2 Keyboard)
- http://www.programmableplanet.com/author.asp?section_id=2142&doc_id=246139 (VGA Display)
- Verilog HDL : Digital Design and Modeling / Joseph Cavanagh - Boca Raton : CRC Press, 2007
- Dynamic RAM - IS42S16400 DatasheetWolfson-WM8731-audio-CODEC Datasheet
- http://it.wikipedia.org/wiki/Pulse-code_modulation.