



FACOLTA' DI INGEGNERIA INFORMATICA - Course 2018 / 2019

DIPARTIMENTO DI INFORMATICA, SCIENZE E INGEGNERIA

SISTEMI DIGITALI M:

THE SNAKE GAME - A VHDL VERSION

Prof. Eugenio Faldella

Realizzato da:

Dott. Luca Cesarano (matr. 0000904861)
Dott. Andrea Croce (matr. 0000904881)

Indice

1 Introduzione	3
1.1 Una breve introduzione a VHDL	3
1.2 Terasic Altera DE1	3
1.2.1 Specifiche tecniche	4
1.3 Introduzione al progetto	5
1.4 Prototipo in Python	6
1.5 La nostra versione	9
2 Organizzazione del progetto	10
2.1 Pulizia della workspace	11
2.2 Cenni sul ROM Editor	11
2.3 Organizzazione del codice	12
3 I dispositivi I/O	13
3.1 Protocollo VGA	13
3.1.1 Il nostro driver	15
3.2 Protocollo PS/2	17
3.3 Display a sette segmenti	21
3.4 LEDs (Light Emitting Diode)	23
4 La logica di gioco	24
4.1 Gli stati del gioco	24
4.2 Il serpente	26
4.3 La gestione del punteggio	29
4.4 La difficoltà di gioco	31
4.5 Posizionamento degli oggetti sullo stage	32
5 Le mappe di gioco	34
5.1 I livelli di gioco	34
5.1.1 Posizionamento dei blocchi	34
5.2 Il caricamento di livelli esterni attraverso il ROM Editor	36
6 Le ROM utilizzate	37
6.1 Generazione del testo	37
6.1.1 Font ROM	37
6.2 Generazione dei bordi	38
6.3 Generazione degli altri elementi	39
6.4 Recupero dei caratteri	40
7 Le animazioni di gioco	42
7.1 Animazione della GUI	42
7.2 Animazione dei frutti	44

8 Conclusione	45
8.1 Sviluppi futuri	45
8.2 Ringraziamenti	45
Riferimenti bibliografici	46

1 Introduzione

Questo paper universitario è stato realizzato al fine di scoprire ed utilizzare la scheda integrata FPGA (Field Programmable Gate Array) Terasic Altera DE1 fornитоci dall'università per la realizzazione di un progetto in VHDL utilizzando l'ambiente di sviluppo integrato Intel Quartus II versione 13.0.

1.1 Una breve introduzione a VHDL

VHDL è un linguaggio appartenente alla categoria HDL (Hardware Description Language). E' utilizzato per la descrizione del comportamento di circuiti elettronici. VHDL è uno standard IEEE (1076) ed è fortemente influenzato dal linguaggio ADA, usato dal dipartimento della difesa statunitense, sia nei concetti che nella sintassi.

Una delle caratteristiche principali di VHDL è la concorrenzialità, ovvero l'abilità di poter eseguire istruzioni VHDL tutte contemporaneamente, differentemente da un classico linguaggio di programmazione, dove le istruzioni sono eseguite sequenzialmente. Esistono tuttavia dei costrutti, chiamati **process**, che descrivono una porzione di codice dal punto di vista algoritmico. In questo modo è possibile scrivere la logica del nostro sistema in maniera più intuitiva. Non è nostro interesse soffermarci troppo sul linguaggio VHDL, ma è possibile consultare i riferimenti in bibliografia per approfondire l'argomento [1].

1.2 Terasic Altera DE1

Il Dev Kit Terasic ALTERA DE1 utilizzato in questo progetto è una scheda embedded FPGA volta alla realizzazione di prototipi dal design avanzato nel campo multimedia, storage e networking. La scheda offre un ampio set di caratteristiche che la rende facilmente utilizzabile da studenti in corsi universitari per un'innumerevole varietà di progetti, così come per lo sviluppo di sofisticati sistemi digitali.

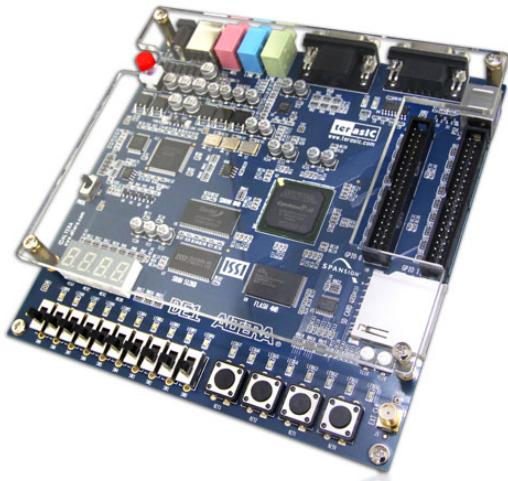


Figura 1: Terasic Altera DE1

1.2.1 Specifiche tecniche

- Altera Cyclone® II 2C20 FPGA device
- Altera Serial Configuration device - EPICS4
- USB Blaster (on board) for programming and user API control; Both JTAG and Active Serial (AS) Programming modes are supported.
- 512-Kbyte SRAM, 8-Mbyte SDRAM, 4-Mbyte Flash memory
- SD Card socket
- 4 pushbutton switches
- 10 toggle switches, 10 user LEDs red, 8 user LEDs green
- 50-MHz oscillator, 27-MHz and 24-MHz oscillator for clock oscillator sources
- CD-quality 24-bit audio CODEC with line-in, line-out and microphone
- VGA DAC (4-bit resistor network) with VGA-out connector
- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- Two 40-in Expansion Headers with Resister Protection
- 7.5V DC Power Supply Connector

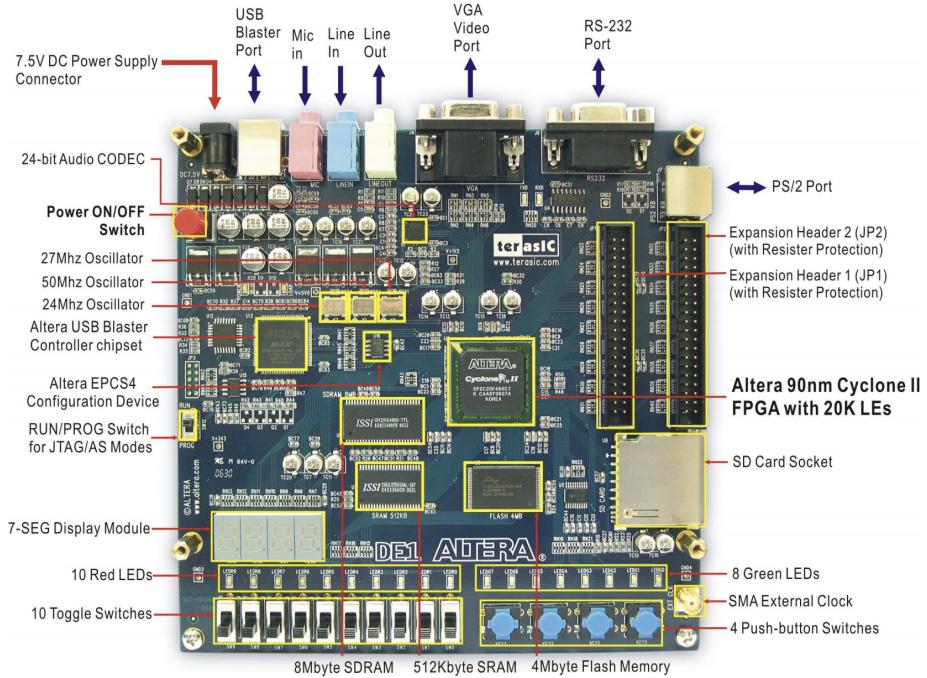


Figura 2: Terasic Altera DE1

1.3 Introduzione al progetto

L'idea di voler sviluppare un videogioco per questo progetto è nata dalla volontà di capire come un sistema multimediale del genere potesse essere implementato su FPGA, non avendo a disposizione i tool di sviluppo che oggi sono presenti sul mercato.

Sviluppare un videogioco in FPGA è simile al programmare un videogioco del passato, come uno di quelli presenti nei cabinati della nostra infanzia.

Il paradigma di sviluppo cambia radicalmente con questa tipologia di sistemi. Oggi, le tecniche di sviluppo prevedono una forte dipendenza da tool chiamati *game engine*, contenenti molteplici funzionalità integrate come il supporto nativo a tantissimi tipi di periferiche I/O. Infatti, essendo la fase di sviluppo molto meno onerosa, le realtà aziendali spendono gran parte delle risorse nella parte pubblicitaria e nell'assuzione di dialoghi e voice actors piuttosto che nello sviluppo vero e proprio del gioco.

Su un FPGA niente dev'essere dato per scontato, in quanto tutto è da implementare da capo. Datasheet alla mano, è necessario scrivere un driver video per la comunicazione con un monitor dotato di porta VGA, un driver per l'interfacciamento di periferiche rimovibili come una tastiera PS/2, ed un altro per l'interfacciamento con il display integrato a sette segmenti. Tutto ciò ci ha messo davanti una sfida entusiasmante che non avremmo mai pensato di affrontare.

Snake è un videogioco arcade famosissimo nato e sviluppato per i primi telefonini, ancor prima dell'avvento degli smartphone. Le regole del gioco sono semplicissime: il giocatore può muovere il serpente in quattro direzioni (su, giù, sinistra, destra) all'interno di un piano bidimensionale. Il serpente deve mangiare il più alto numero di frutti possibili in modo da incrementare il suo punteggio, senza sbattere contro se stesso o contro gli ostacoli, in quanto porterebbero ad un *game over*. Ogni volta che un frutto viene ingerito il serpente aumenta la sua lunghezza, aumentando la probabilità di collisioni con il mondo circostante. È possibile inoltre coprire lunghe distanze in breve tempo attraverso i punti di teletrasporto inseriti ai bordi dello stage che permettono al giocatore di scomparire e ricomparire nella parte laterale opposta dello stage.

1.4 Prototipo in Python



Figura 3: Prototipo in Python

La realizzazione di un prototipo è assolutamente necessaria per schiarirsi le idee prima di affrontare una qualunque progettazione in VHDL. Nel nostro caso, capire come funzionasse parte della logica di gioco come la crescita del serpente, le collisioni del serpente con se stesso ed i teletrasporti ai bordi è stato essenziale ai fini del compimento del progetto in VHDL.

Alleghiamo il codice Python:

```
import curses
from curses import KEY_RIGHT, KEY_LEFT, KEY_UP, KEY_DOWN
from random import randint

# Inizializzazione della schermata
curses.initscr()
windows = curses.newwin(20, 60, 0, 0)
windows.keypad(1)
curses.noecho()
curses.curs_set(0)
windows.border(0)
windows.nodelay(1)

# Inizializzazione dei valori
key = KEY_RIGHT
score = 0

# Coordinate iniziali del serpente e della mela
snake = [[4,10], [4,9], [4,8]]
apple = [10,20]

# Mostra le mele
windows.addch(apple[0], apple[1], '*')

# Finche' ESC non viene premuto
while key != 27:
    windows.border(0)
    #Label Snake e Score
    windows.addstr(0, 2, 'Score : ' + str(score) + ' ')
    windows.addstr(0, 27, ' SNAKE ')
    #Aumenta velocita' del serpente a seconda della sua lunghezza
    windows.timeout(150 - (len(snake)/5 + len(snake)/10) % 120)

    # Previous Key premuto
    prevKey = key
    event = windows.getch()
    key = key if event == -1 else event

    # Se viene premuto un tasto non mappato
    if key not in [KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN, 27]: key = prevKey
```

```

# Calcola le nuove coordinate della testa del serpente
snake.insert(0, [snake[0][0] + (key == KEY_DOWN and 1) + (key == KEY_UP and -1),
                snake[0][1] + (key == KEY_LEFT and -1) + (key == KEY_RIGHT and 1)])

# Se il serpente tocca i bordi, compare sul lato opposto
if snake[0][0] == 0: snake[0][0] = 18
if snake[0][1] == 0: snake[0][1] = 58
if snake[0][0] == 19: snake[0][0] = 1
if snake[0][1] == 59: snake[0][1] = 1

# Se il serpente si tocca da solo e' GAME OVER
if snake[0] in snake[1:]: break

# Serpente mangia la mela
if snake[0] == apple:
    apple = []
    score += 1
    while apple == []:
        # Spawn di una nuova mela
        apple = [randint(1, 18), randint(1, 58)]
        if apple in snake: apple = []
        windows.addch(apple[0], apple[1], '*')
else:
    last = snake.pop()
    windows.addch(last[0], last[1], ' ')
    windows.addch(snake[0][0], snake[0][1], '#')

curses.endwin()

```

1.5 La nostra versione

La nostra versione del gioco, diversamente da quella originale, prevede l'aggiunta di frutti speciali chiamati *power up* che permettono al giocatore di guadagnare più punti e di fornirgli una vita extra al loro ingerimento.

Per rendere il sistema flessibile e rigiocabile, abbiamo sviluppato appositamente per questo progetto, un semplice *ROM Editor* che ha lo scopo di caricare nuovi livelli in gioco, staticamente. In questo modo, con un po' di pratica, è possibile sviluppare nuovi livelli di gioco e caricarli all'interno di esso, rendendo l'esperienza personalizzata e divertente, caratteristica essenziale per un buon videogioco arcade.

Per avere un'idea più chiara del progetto, abbiamo realizzato un breve video disponibile per la consultazione in bibliografia che mostra il contenuto del nostro progetto finalizzato [2].



Figura 4: Schermata del gioco

2 Organizzazione del progetto

Siamo fermamente convinti che partire da un progetto e da un codice ordinato porti grossi risparmi di tempo durante la fase di implementazione del progetto, ed è per questo che alcuni strumenti per l'organizzazione e la pulizia del codice sono stati essenziali per questo scopo.

Prima tra tutti, presentiamo la gerarchia della nostra *workspace*:

the_snake_game

- I- db
- I- incremental_db
- I- **the_snake_game.qpf**
- I- **the_snake_game.qsf**
- I- **the_snake_game.qws**
- I- output_files
- I- resources
- I- romEditor
- I- vhdl_files
- I- **cleanDirectory.bat**
- I- **DE1_pin_assignments.csv**

I primi cinque files sono richiesti da Quartus per il funzionamento dell'ambiente di sviluppo integrato.

output_files contiene i file relativi al deploy del progetto. Per il deploy tramite JTAG si utilizza il file .sof e l'FPGA impostato in **RUN Mode**.

resources rappresenta le risorse di progetto, tra cui appunti e documentazioni per la programmazione dell'FPGA.

vhdl_files contiene la nostra divisione in *folders* del progetto. Una divisione su più livelli permette di gestire modularmente il nostro progetto; ogni modulo rappresenta una parte di progetto diversa.¹

Di seguito alleghiamo la gerarchia della cartella **vhdl_files**:

vhdl_files

- I- applLogic
- I- libs
- I- logic
 - I- gui
- I- peripherals
- I- roms
- I- utils

¹Per evitare ambiguità, precisiamo che il termine *package* viene utilizzato in VHDL per indicare una collezione accessibile globalmente di funzioni, costanti, e tipi di dati.

applLogic contiene il file principale *the_snake_game.vhd*, che connette tutte le componenti di progetto ed avvia l'intero circuito descritto.

libs contiene il file *package*, contenente costanti e funzioni comuni a tutto il progetto; al suo interno sono anche presenti riferimenti ai parametri per il VGA e costanti che, se modificate opportunamente, possono cambiare la logica di gioco, il suo aspetto grafico ed altre funzionalità.

logic contiene tutti gli elementi relativi alla vera e propria logica di gioco, cominciando dall'automa a stati finiti che definisce il comportamento dell'intero sistema. Vi è presente anche la gestione degli oggetti, la gestione delle vite, ed una subfolder **gui** che contiene gli elementi grafici di gioco come le mappe, il serpente e le scritte.

2.1 Pulizia della workspace

È noto che l'ambiente di sviluppo integrato Quartus generi un'ampia quantità di file temporanei che invadono l'ambiente *workspace*. In aggiunta, l'editor di Quartus crea un backup per ogni singola modifica effettuata su un file. Da qui è nata l'esigenza di scrivere un piccolo script chiamato **cleanDirectory.bat** per la pulizia della workspace.

2.2 Cenni sul ROM Editor

Un fattore molto importante affinché un gioco arcade piaccia all'utenza è sicuramente la sua flessibilità. Come già accennato prima, poter rigiocare lo stesso gioco ma in salse diverse rende la sfida sempre più entusiasmante ed assicura che il prodotto rimanga giovane nel tempo. Il ROM Editor è un programma scritto in Python che fornisce all'utente la possibilità di inserire nuovi livelli nel gioco, possibilmente adatti al proprio livello di bravura.

romEditor contiene i file relativi al ROM Editor, compreso lo script vero e proprio (consultabile liberamente) ed i set sono modificabili per la creazione di livelli personalizzati.

Il ROM Editor effettua un backup automatico degli stage rimpiazzati, in modo tale da poterli facilmente ripristinare in caso di problematiche.



Figura 5: Interfaccia del ROM Editor

2.3 Organizzazione del codice

La modularità delle componenti di gioco rispetta la strategia *divide et impera* decomponendo il sistema in unità leggibili ed organizzabili facilmente.

Avere un codice sempre chiaro e pulito evita grossi problemi di manutenzione e soprattutto, aiuta un eventuale manutentore esterno nella lettura del codice. Proprio per questo, il nostro codice cerca di encapsulare il più possibile gli insegnamenti impartiti durante il corso.

Un codice troppo espressivo inoltre può risultare eccessivamente lungo ed è per questo che bisogna trovare un *trade off* per l'ottenimento di un codice che sia chiaro, leggibile ed efficiente.

3 I dispositivi I/O

3.1 Protocollo VGA

L'interfaccia VGA (Video Graphics Array) si presenta come un connettore ad alta densità a 15 pin. Solo 5 di essi sono utilizzati per la comunicazione del segnale video. Precisamente, i segnali adibiti alla visualizzazione sono i primi tre pin a 4 bit, relativi ai colori (rosso, verde e blu); i pin 13 e 14 sono invece i segnali di sincronismo orizzontale e verticale.

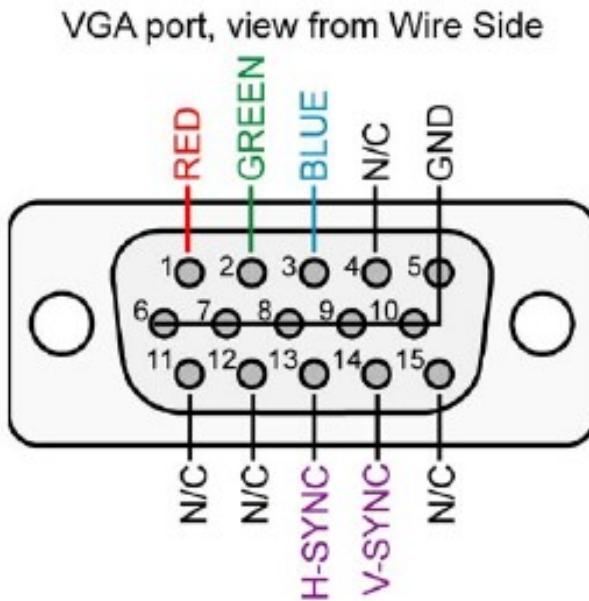


Figura 6: Descrizione dei pin VGA

L'immagine è ottenuta tramite una scansione orizzontale da sinistra a destra e verticale dall'alto al basso. Alla fine di ogni linea viene emesso un impulso di sincronismo orizzontale, mentre alla fine di ogni frame viene emesso un impulso di sincronismo verticale attraverso determinate temporizzazioni che verranno spiegate a breve.

Al momento per il nostro progetto abbiamo associato un solo bit a canale ottenendo così 8 combinazioni di colori diverse così come riportate in tabella.

La banda caratteristica dei segnali viene calcolata attraverso la seguente formula:

$$B = N_{pixel_h} * N_{pixel_v} * N_{frame_sec} * t_{retrace} \quad (1)$$

La frequenza ottenuta è chiamata *pixel clock* ed indica la velocità con cui i dati relativi ai segnali di colore devono essere emessi in uscita.

Le temporizzazioni orizzontali e verticali hanno una scala temporale diversa in quanto la scansione verticale è tipicamente effettuata solamente a fronte di una linea scansionata, risultando molto più lenta di quella orizzontale. Dalla figura in allegato è possibile distinguere un'area attiva ed un'area inattiva. Al di fuori della zona attiva, la zona inattiva è necessaria affinché la sincronizzazione abbia successo; essa contiene il *front porch*, il *back porch* ed il *sync pulse*.

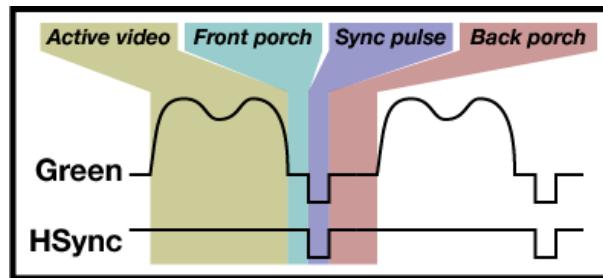


Figura 7: Grafico illustrante la temporizzazione VGA

Inizialmente i segnali RGB devono essere necessariamente spenti affinché il monitor possa sincronizzarsi. Teniamo presente che, sebbene molti dispositivi riescano ad agganciarsi ai tempi utilizzati anche se non sono molto precisi, è altamente consigliata l'ottenimento di una buona precisione attraverso un'operazione di *tuning dei parametri VGA*, adeguata in modo da non generare artefatti grafici a video.

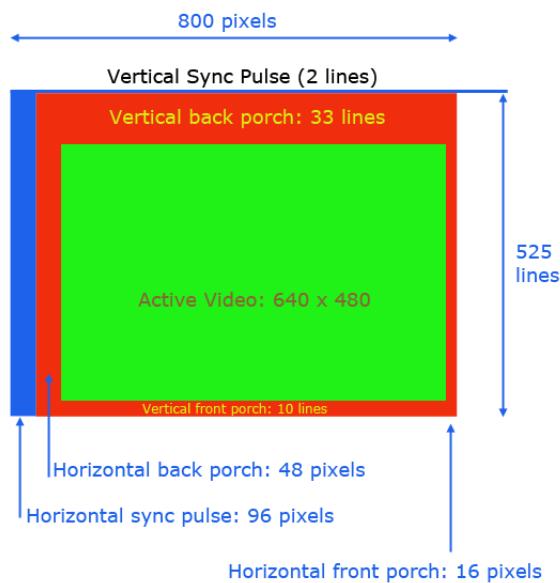


Figura 8: Grafico illustrante la parte attiva ed inattiva di una schermata VGA

3.1.1 Il nostro driver

Per l'implementazione del driver VGA abbiamo utilizzato lo standard $640 \times 480 @ 60 Hz$. Per la realizzazione di tale scelta è necessario che i parametri descritti nel paragrafo precedente siano impostati attraverso la seguente tabella.

Formato	Pixel Clock	H-Resolution	H-Front Porch	H-Sync Pulse
640x480 @60Hz	25.175 MHz	640	16	96
H-Back Porch	V-Resolution	V-Front Porch	V-Sync Pulse	V-Back Porch
48	480	11	2	31

Tabella 1: Tabella dei parametri

```
-- status
horizontal_end <= -- end of horizontal counter
'1' when horizontal_counter_register =
(HORIZONTAL_DISPLAY + HORIZONTAL_FRONT_PORCH
+ HORIZONTAL_BACK_PORCH + HORIZONTAL_RTRACE - 1) else '0'; --799

vertical_end <= -- end of vertical counter
'1' when vertical_counter_register =
(VERTICAL_DISPLAY + VERTICAL_FRONT_PORCH
+ VERTICAL_BACK_PORCH + VERTICAL_RTRACE - 1) else '0'; --524
```

Quindi in totale abbiamo una quantità di pixel orizzontale pari a 800 mentre una quantità di pixel verticali a 525, area inattiva inclusa.

La frequenza di $60 Hz$ al secondo è consigliata affinché l'esperienza di gioco risulti fluida e scorrevole.

Per la scrittura del component in VHDL ci siamo inspirati all'implementazione hardware consultata sul libro *FPGA Prototyping by VHDL Examples: Xilinx Spartan™-3 Version Chu* [3].

La scansione VGA è fatta *pixel by pixel*, ovvero viene scansionata linea per linea la nostra matrice di LEDs.

La frequenza del nostro clock è di 50 MHz tuttavia la frequenza richiesta dal protocollo è di 25 MHz dunque nel codice è previsto l'utilizzo di un segnale chiamato *pixel tick* che dimezza la frequenza di clock di partenza, e coordinare le attività per la generazione di pixel.

Vengono usate inoltre *vertical_end* e *horizontal_end* per controllare il completamento dello scanning orizzontale e verticale.

Infine, per evitare interferenze e glitch grafici, così come menzionato precedentemente, il canale è bufferizzato nel seguente modo:

```
-- horizontal and vertical sync, buffered to avoid glitch
horizontal_sync_next <= '1' when (horizontal_counter_register >=
(HORIZONTAL_DISPLAY + HORIZONTAL_FRONT_PORCH)) --656
and (horizontal_counter_register <=
(HORIZONTAL_DISPLAY + HORIZONTAL_FRONT_PORCH
+ HORIZONTAL_RETRACE - 1)) else '0'; --751

vertical_sync_next <= '1' when (vertical_counter_register >=
(VERTICAL_DISPLAY + VERTICAL_FRONT_PORCH)) --490
and (vertical_counter_register <=
(VERTICAL_DISPLAY + VERTICAL_FRONT_PORCH
+ VERTICAL_RETRACE - 1)) else '0'; --491
```

Per quanto riguarda la manipolazione dei colori che sono stati assegnati alle varie componenti del gioco, ci rifacciamo alla seguente tabella:

VGA Red	VGA Green	VGA Blue	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Figura 9: Tabella colori a 3 bit

È possibile impostare il colore di un elemento a video attraverso una stringa a 3 bit. Per esempio, se volessimo colorare di blu la testa del serpente dovremmo impostare la costante relativa alla testa del serpente con il valore *001*.

```
constant HEAD_COLOR: std_logic_vector(2 downto 0):="010";
constant BODY_COLOR: std_logic_vector(2 downto 0):="010";
```

Questo discorso si estende a tutti gli elementi grafici che ritroviamo nel nostro sistema.

3.2 Protocollo PS/2

Il protocollo PS/2 prevede una comunicazione che avviene tra tastiera ed host tramite un connettore *MINI DIN maschio a 6 pin*, riportato per la consultazione in figura.

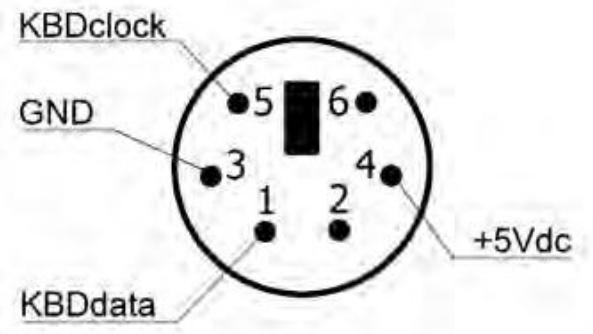


Figura 10: Illustrazione dei PIN nel protocollo PS/2

Tipicamente la tastiera PS/2 invia all'host (la nostra board) una sequenza di bit (*scan codes*) indicante il tasto premuto. Ogni volta che il tasto premuto viene rilasciato viene inviato un *break code* seguito dal codice del tasto rilasciato.

ESC 76	F1 05	F2 06	F3 04	F4 06	F5 03	F6 0B	F7 83	F8 0A	F9 01	F10 09	F11 78	F12 07	↑ EO75
~ 0B 16	! 1! 1E	2 @ 26	3 # 25	4 \$ 25	5 % 2E	6 ^ 36	7 & 3D	8 * 3E	9 (0) 46	- 45	= + 45	Back Space 4B 55	← 66
TAB 0D	Q 15	W 1D	E 24	R 2D	T 2C	Y 35	U 3C	I 43	O 44	P 4D	{ { } } 54	5B 5A	 5D
Caps Lock 58	A 1C	S 1B	D 23	F 2B	G 34	H 33	J 3B	K 42	L 4B	:: 4C	" " 52	Enter 4D 5A	EO6B
Shift △ 12	Z 1Z	X 22	C 21	V 2A	B 32	N 31	M 3A	, < 41	> . 49	/ ? 4A	Shift 59	EO72	↓
Ctrl 14	Alt 11		Space 29		Alt EO11		Ctrl EO14						

Figura 11: Gli scan codes relativi al protocollo PS/2

La comunicazione tra host e tastiera è bidirezionale (l'host è prioritario). Per la ricezione dei dati da parte della tastiera, è necessario che il segnale di *Clock* ed il segnale *Data* siano entrambi abilitati, chiamata condizione di *Idle*. Questa è l'unica condizione nella quale la tastiera può trasmettere dati. Se l'Host non abilita il segnale di *Clock*, tutti i segnali inviati dalla tastiera vengono bufferizzati e l'invio viene ripreso solamente dopo l'abilitazione del *Clock*.

Il dispositivo genera sempre il segnale di *Clock*. Se l'host vuole inviare dati, deve abbassare il *Clock* a *low*. L'host, a quel punto, porta la linea *data* a *low* e rilascia il *Clock*. Questo stato è chiamato *Request to Send*.

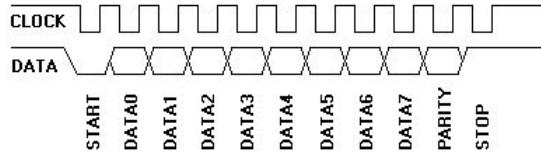


Figura 12: Illustrazione degli 11 bit del protocollo PS/2

La trasmissione dei dati avviene su 11 bit. Il primo bit è chiamato *start bit* ed è seguito da 8 bit dedicati ai dati. Il bit 9 è un *parity bit* e viene seguito dal bit 10 chiamato *stop bit* indicante la fine della trasmissione.

Le interazioni tra host e tastiera avvengono da parte dell'host che porta a 0 la linea dati. Il clock è mantenuto a livello basso per evitare una situazione in cui la tastiera inizia la trasmissione dati nello stesso momento dell'host.

L'invio dei 9 bit (8 di dati più 1 di parità) porta un innalzamento della linea dati allo stato *Idle* per un ciclo di clock. La tastiera risponde con un segnale di ACK mantenendo la linea dati a 0 per il successivo ciclo di clock. Se la linea dati non viene messa a *Idle* la tastiera invierà continuamente il segnale di *clock* fino al verificarsi della condizione.

Tasto	Code	Comando
UP	E075	MOVE PLAYER UP
DOWN	E072	MOVE PLAYER DOWN
LEFT	E06B	MOVE PLAYER LEFT
RIGHT	E074	MOVE PLAYER RIGHT
ENTER	5A	SELECT
SPACE	29	PAUSE / RESUME
ESC	76	RESET

Tabella 2: Tabella dei comandi

Alleghiamo un piccolo snippet relativo al processo per l'individuazione del tasto premuto. Il tasto premuto viene inserito in un buffer che viene successivamente trasmesso nel vettore *buttons*.

```
process (ps2rx)

begin
if ps2rx'event and ps2rx = '0' then

    -- ENTER BUTTON
    if ps2rx.d1 = X"5A" then
        if ps2rx.d2 = X"F0" then
            buttons_temp(4) <= '0';
        else
            buttons_temp(4) <= '1';
        end if;
    end if;

    -- ESC BUTTON
    if ps2rx.d1 = X"76" then
        if ps2rx.d2 = X"F0" then
            buttons_temp(6) <= '0';
        else
            buttons_temp(6) <= '1';
        end if;
    end if;

end if;
end process;

BUTTONS <= buttons_temp;
```

Per capire un po' meglio il protocollo, elenchiamo una serie di passi indicanti un *use case* che l'host deve seguire per inviare dati ad un dispositivo PS/2.

1. Innalzare la linea *Clock* per almeno 100 microsecondi;
2. Abbassare la linea *Data*;
3. Rilasciare la linea *Clock*;
4. Aspettare che il dispositivo abbassi la linea *Clock*;
5. Impostare la linea *Data* all'invio del primo bit;
6. Aspettare che il dispositivo innalzi la linea *Clock*;
7. Aspettare che il dispositivo abbassi la linea *Clock*;
8. Ripetere gli step 5-7 per gli altri 7 bit ed il parity bit;
9. Rilasciare la linea *Data*;
10. Aspettare che il dispositivo abbassi *Data*;
11. Aspettare che il dispositivo abbassi *Clock*;
12. Aspettare che il dispositivo rilasci *Data* e *Clock*.

3.3 Display a sette segmenti

Il modulo *score.vhd* si occupa di visualizzare sul display a 7 segmenti presente sullo schedino il punteggio di gioco corrente per un'interazione maggiore con l'hardware dell'FPGA. Le 4 cifre in uscita non saranno mostrate solamente in hardware ma anche sulla GUI contenuta sul display.

Per la modifica dei valori sul display hardware è necessario conoscere la tabella di verità di un display a 7 segmenti. Ogni segmento corrisponde ad un bit diverso che, se posto su *low*, rimane acceso. Lo stato *high* serve allo spegnimento del segmento.

Abbiamo a disposizione quattro cifre, ogni cifra è una stringa a 7 caratteri dove ogni elemento è rappresentato da un segmento diverso.

Per esempio, se volessimo indicare il numero 0 sullo schedino in hardware sarebbe sufficiente impostare su *high* il settimo segmento del display, ovvero l'ultimo, con stringa risultante *0000001*.

TRUTH TABLE FOR 7-SEGMENT DECODER

X	a	b	c	d	e	f	g
0 (0000)_b	0	0	0	0	0	0	1
1 (0001)_b	1	0	0	1	1	1	1
2 (0010)_b	0	0	1	0	0	1	0
3 (0011)_b	0	0	0	0	1	1	0
4 (0100)_b	1	0	0	1	1	0	0
5 (0101)_b	0	1	0	0	1	0	0
6 (0110)_b	0	1	0	0	0	0	0
7 (0111)_b	0	0	0	1	1	1	1
8 (1000)_b	0	0	0	0	0	0	0
9 (1001)_b	0	0	0	0	1	0	0
A (1010)_b	0	0	0	1	0	0	0
B (1011)_b	1	1	0	0	0	0	0
C (1100)_b	0	1	1	0	0	0	1

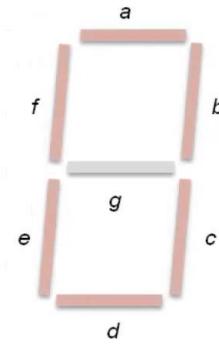


Figura 13: Tabella di verità di un display a 7 segmenti

```

-- Setting the score on the embedded four 7-segments displays.
with digit_0_register select
    HEX0 <=
        "1000000" when "0000", --0
        "1111001" when "0001", --1
        "0100100" when "0010", --2
        "0110000" when "0011", --3
        "0011001" when "0100", --4
        "0010010" when "0101", --5
        "0000010" when "0110", --6
        "1111000" when "0111", --7
        "0000000" when "1000", --8
        "0010000" when "1001", --9
        "1111111" when others;

with digit_1_register select
    HEX1 <=
        "1000000" when "0000", --0
        "1111001" when "0001", --1
        "0100100" when "0010", --2
        "0110000" when "0011", --3
        "0011001" when "0100", --4
        "0010010" when "0101", --5
        "0000010" when "0110", --6
        "1111000" when "0111", --7
        "0000000" when "1000", --8
        "0010000" when "1001", --9
        "1111111" when others;

```

3.4 LEDs (Light Emitting Diode)

I led verdi integrati sull'FPGA sono stati utilizzati per indicare lo stato corrente del sistema. Gli stati verranno poi discussi approfonditamente nella sezione relativa all'automa stati finiti utilizzato per questo sistema.

STATO	LED ON
GAME OVER	: LEDG0
STAGE CLEAR	: LEDG1
LIFE LOST	: LEDG2
LEVEL UP	: LEDG3
PAUSE	: LEDG4
PLAY	: LEDG5
SELECT	: LEDG6
START	: LEDG7

```
LEDG <= state_indicator_var
state_indicator_var <=
    "10000000" when state_register=new_game else
    "01000000" when state_register=stage_select else
    "00100000" when state_register=playing else
    "00010000" when state_register=paused else
    "00001000" when state_register=levelup else
    "00000100" when state_register=new_life else
    "00000010" when state_register=stageclear else
    "00000001" when state_register=game_over else
    "00000000";
```

4 La logica di gioco

4.1 Gli stati del gioco

Un automa a stati finiti (*abbreviato in ASF*) è un modello matematico computazionale che descrive formalmente il comportamento di un sistema. Un ASF è dotato di stato, e le transizioni tra stati avvengono tramite sollecitazione da input esterni. In un preciso istante il sistema è rappresentato da uno stato. Lo *stato iniziale* indica il primo stato in cui si trova l'automa.

Di seguito è riportato il diagramma dell'automa a stati finiti:

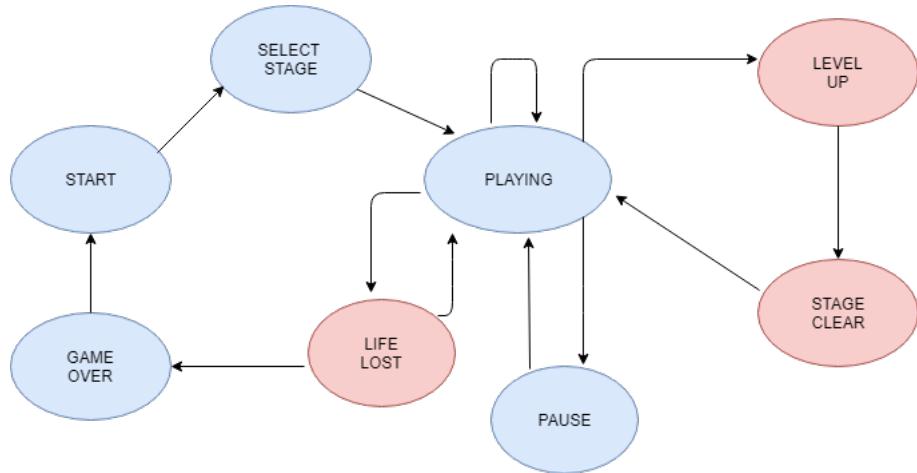


Figura 14: Automa a Stati Finiti del nostro sistema

Gli stati in rosso sono *stati temporanei*, ovvero stati che, dopo un certo tempo di computazione, effettuano spontaneamente una transizione di stato.

Nel diagramma sono stati appositamente omessi i nomi delle azioni che innescano una transizione di stato in modo che l'automa si astragga dal sistema di input utilizzato. Nel nostro sistema, il sistema di input è il protocollo PS/2. La mappatura tra tasti ed azioni è consultabile nella sezione relativa all'interfacciamento con periferiche PS/2.

Il file che racchiude la logica del nostro automa a stati finiti è chiamato *finite_state_machine.vhd*. Questo modulo rappresenta la parte di codice più importante di tutto il progetto, in quanto rappresentante della logica pura dell'ASF. Tutte le condizioni che portano ad un cambiamento di stato sono indicate in questo file, come la condizione *hit*, indicante che il serpente è entrato in collisione con un elemento di gioco.

```

-- If you're playing and you press space the game pauses
when playing =>
    if button_space='1' then
        state_next <= paused;
-- If you get hit the new life state gets triggered
    elsif hit='1' then
        state_next <= new_life;
-- If the stage gets cleared the stageclear gets triggered
    elsif STAGE_CLEAR='1' then
        state_next <= stageclear;
    elsif LEVEL_UP='1' then
-- If level up is high we need to change level
        state_next <= levelup;
    end if;

```

In questo modulo sono mappati i tasti fisici della tastiera ad azioni di gioco.

```

-- Assignment between BUTTONS and std_logic dedicated to them.
-- BUTTONS are taken from the ps/2 component.

```

```

button_down      <= BUTTONS(0);
button_up        <= BUTTONS(1);
button_left      <= BUTTONS(2);
button_right     <= BUTTONS(3);
button_enter     <= BUTTONS(4);
button_space     <= BUTTONS(5);

```

4.2 Il serpente

La progettazione del serpente è una delle parti più articolate del progetto. La parte di design è abbastanza semplice, e verrà discussa nei capitoli successivi.

In una fase di inizializzazione viene impostata la posizione del serpente sullo stage. Il posizionamento è gestito da un processo che, quando innescato il meccanismo di *reset*, prevede il riposizionamento della testa del serpente (che, ad inizio gioco, è il serpente stesso) nella posizione iniziale.

```
process(CLOCK, RESTART)
begin
    if CLOCK'event and CLOCK='1' then
        if RESTART(1)='1' or RESTART(0)='1' then
            head_x_register      <= to_unsigned(320,10);
            head_y_register      <= to_unsigned(240,10);
        else
            head_x_register      <= head_x_next;
            head_y_register      <= head_y_next;
            direction_register   <= direction_next;
        end if;
    end if;
end process;
```

Il serpente dev'essere visto come un buffer di elementi, tutti uguali tra loro. Il primo elemento è la testa del serpente (*head*), attaccata alla testa vi sono tanti elementi del corpo (*body*) quanti frutti il serpente mangia. Ogni elemento segue ricorsivamente il movimento dell'elemento precedente; così che, se la testa si muove a destra, l'elemento *body*₁ si muove come l'elemento *head*, l'elemento *body*₂ si muove come l'elemento *body*₁ e così per tutti gli altri elementi in coda.

All'ingestione di un frutto il serpente incrementa la sua lunghezza di un blocco; l'effetto è ottenuto attraverso l'inserimento in coda di un elemento di tipo *body* (geometricamente risulterà alla posizione in cui si è trovato l'ultimo elemento *body* del serpente).

Come facciamo a capire se il serpente ha effettivamente ingerito un frutto? Questo quesito introduce la gestione delle collisioni. Una collisione prevede un controllo della posizione di un oggetto A con quello di un oggetto B. Se la distanza tra i due oggetti è nulla, ovvero gli oggetti si trovano sullo stesso punto, allora si dice che gli oggetti *collidono*.

$$d(x, y) = (x_0, y_0) - (x_1, y_1) = 0 \quad (2)$$

La collisione *serpente-frutto* è la più semplice perché basta controllare il posizionamento della testa del serpente con il frutto.

```
-- item ate
item_ate_var <= '1' when ITEM_X=head_x_register and ITEM_Y=head_y_register else '0';
```

La variabile assume così valore alto, che indica che il punteggio dev'essere incrementato. Il funzionamento delle collisioni con gli altri oggetti è analogo. Elenchiamo una tabella di tutte le combinazioni e le loro reazioni presenti nel gioco.

ITEM 1	ITEM 2	EFFECT
HEAD	BODY	LIFE LOST
HEAD	BLOCK	LIFE LOST
HEAD	FRUIT	SCORE UP
HEAD	SP. FRUIT	EXTRA LIFE
HEAD	BORDERS	TELEPORT OPPOSITE SIDE
ITEM	BLOCK	PLACE ITEM AGAIN
ITEM	SNAKE	PLACE ITEM AGAIN
ITEM	ITEM	PLACE ITEM AGAIN

Tabella 3: Tabella delle collisioni

Il teletrasporto del serpente, ovvero la facoltà di potersi spostare istantaneamente da un bordo dello stage ad un altro, avviene con lo stesso sistema di collisioni. Si controlla se il serpente ha raggiunto una determinata posizione (la posizione del bordo) e lo si riposiziona sul bordo appartenente al lato opposto. In questo modo, è possibile coprire lunghe distanze in breve tempo, facilitando l'esperienza di gioco ed il raccoglimento dei frutti speciali, che possono essere raccolti solo in un breve lasso di tempo.



Figura 15: Teletrasporto del Serpente

Quindi, se il serpente tocca il bordo sinistro dello stage, apparirà sul bordo destro. Il registro contenente la posizione della testa, *head_x_register*, verrà riposizionato alla posizione di destinazione; da notare che non c'è bisogno di modificare tutti gli elementi del corpo, perché tramite *effetto domino* automaticamente l'elemento *n* seguirà il movimento dell'elemento *n - 1* e lo stesso avviene per gli elementi rimanenti.

```
-- TELEPORT FROM LEFT TO RIGHT
case head_x_register is
    when "0000000000" =>
        head_x_next <= to_unsigned(MAX_X-BLOCK_SIZE,10);
    when others =>
        head_x_next <= head_x_next - BLOCK_SIZE;
end case;
```

Lo spostamento del serpente sullo stage avviene tramite uno scorrimento dell'elemento in testa (e, ricorsivamente, degli altri elementi) sulla scacchiera di pixel a disposizione. Ci avvaliamo di un registro *next* che calcola la posizione in base alla direzione del serpente. La direzione è dettata dal pulsante premuto sulla tastiera.

```
-- snake control
direction_next <=
    "00" when BUTTONS = "0001" else
    "01" when BUTTONS = "0010" else
    "10" when BUTTONS = "0100" else
    "11" when BUTTONS = "1000" else
    direction_register;
```

Un ultimo aspetto da menzionare sono i vincoli del serpente, che può muoversi solamente di 90° a destra o 90° a sinistra, le altre direzioni sono vietate.

```
-- Positions Check, if you MOVE in a direction you can ONLY MOVE +-90°
case direction_register is
    when "00" =>
        if direction_temp /= "01" then direction_temp := "00";
        end if;
    when "01" =>
        if direction_temp /= "00" then direction_temp := "01";
        end if;
    when "10" =>
        if direction_temp /= "11" then direction_temp := "10";
        end if;
    when others =>
        if direction_temp /= "10" then direction_temp := "11";
        end if;
end case;
```

4.3 La gestione del punteggio

Per il calcolo del punteggio ci affidiamo a due componenti. La prima, *score*, ha il compito di decidere, a seconda del livello raggiunto e del power up ingerito, quanto punteggio assegnare. Il punteggio assegnato viene comunicato al componente *four-digit-counter*, un contatore a 4 bit che conta fino a 9999, per poi ripartire da 0.²

L'algoritmo di attribuzione del punteggio è un semplice contatore che valuta se la condizione *cibo ingerito* è vera; in quel caso viene alzato un flag *add*, indicante che dev'essere aumentato lo score in uscita e viene contato il punteggio da attribuire attraverso la formula

$$score_t = score_{t-1} + level + item_{type} \quad (3)$$

```

begin
    if CLOCK'event and CLOCK='1' then
        if ITEM_ATE = '1' then
            add := '1';
        end if;

        if add = '1' then
            if counter = LEVEL + ITEM_TYPE then
                add := '0';
                counter := 0;
            else
                counter := counter + 1;
            end if;
        end if;
    -- This is the value that goes as input in 4_digit_counter to
    -- handle the score represented by 4 digits (counter 9999)
    digit_increase <= add;
    end if;
end process;
```

Come accennato poc'anzi, il flag *add* viene messo in ingresso al *four-digit-counter* per l'aumento del conteggio. Il counter si occupa di gestire il contatore a 4 bit e di aumentare il punteggio laddove gli viene comunicato un *digit_increase* in ingresso.

Pian piano che il giocatore aumenta il suo punteggio, un sistema di *leveling* interno ha la responsabilità di proporre all'utente nuovi stages automaticamente. Quando accade, compare a video una scritta *Stage Clear*, così come indicato nell'automa a stati finiti ad inizio capitolo.

²Per il sistema di attribuzione del punteggio utilizzato, è davvero difficile arrivare ad un punteggio di 9999, se non impossibile. Questo riduce a pressoché zero il rischio di *overflow* del punteggio, per cui non sono stati effettuati troppi controlli affinché il contatore non vada in *overflow*.

```

begin
    if CLOCK'event and CLOCK='1' then
        level_up_temp      := '0';
        stage_clear_temp   := '0';

        if RESTART(1) = '1' then
            level_temp       := 1;
            stage_select_tmp := STAGE_SELECT_IN;
            counter_next_level := 0;

        elsif RESTART(0) = '1' then
            counter_next_level := 0;

        elsif ITEM_ATE = '1' then

            if counter_next_level = 2 * level_temp then level_temp := level_temp + 1;

            if level_temp = 6 then
                if stage_select_tmp < 2 then
                    stage_select_tmp := stage_select_tmp + 1;
                end if;
                stage_clear_temp := '1';
                level_temp       := 1;
            end if;
            level_up_temp      := '1';
            counter_next_level := 0;
        else
            counter_next_level := counter_next_level + 1;
        end if;
    end if;

    LEVEL           <= level_temp;
    LEVEL_UP        <= level_up_temp;
    STAGE_CLEAR     <= stage_clear_temp;
    STAGE_SELECT_OUT <= stage_select_tmp;

end if;
end process;

```

4.4 La difficoltà di gioco

Vi è una proporzionalità diretta tra la lunghezza del serpente e la difficoltà del gioco; maggiore è il numero degli elementi che il serpente ingerisce e maggiore sarà la sua lunghezza, incrementando la probabilità di collisione con gli oggetti circostanti.

Un contributo relativo all'aumento o la diminuzione della difficoltà in gioco è il quantitativo di vite a disposizione del giocatore. Una vita extra ti permette di effettuare una nuova partita in caso di perdita senza azzerare il punteggio raggiunto, consentendoti di raggiungere punteggi sempre più elevati.

```
-- Snake got HIT, we decrease a life.  
elsif HIT = '1' then  
    if counter = 2 then  
        lives_temp := lives_temp - 1;  
        counter := 0;  
    else  
        counter := counter + 1;  
    end if;
```

Ricapitolando, in caso di collisione contro il *body* o i *block*, viene decrementata una vita. Se tutte le vite si esauriscono, si entra nello stato di *game over* ed il punteggio viene azzerato.

Per facilitare le cose, in gioco è presente un frutto bonus che regala al giocatore una vita extra, se raccolto in tempo.

```
-- Item has been eaten? If it's type 3 then add a life.  
elsif ITEM_ATE = '1' and ITEM_TYPE = 3 then  
    if counter = 2 then  
        if lives_temp < 7 then  
            lives_temp := lives_temp + 1;  
        end if;  
        counter := 0;  
    else  
        counter := counter+1;  
    end if;
```

4.5 Posizionamento degli oggetti sullo stage

L'algoritmo di generazione dei frutti sullo stage è incredibilmente importante; esso simula la generazione pseudo-casuale di numeri rappresentanti le coordinate di gioco.

Vengono utilizzati tre contatori, i quali contano sempre finché il gioco si trova nello stato di *playing*. Il primo contatore rappresenta la generazione della coordinata x del frutto da istanziare; il secondo rappresenta quella della coordinata y mentre il terzo rappresenta la generazione del tipo di frutto da istanziare in gioco; come avevamo già accennato, ci sono quattro tipi di frutti utilizzabili in gioco.

Per ottenere le coordinate dal valore del contatore:

$$(x, y, z) = (counter_x * 16, counter_y * 16, counter_z * 16) \quad (4)$$

Il dominio di appartenenza dei frutti deve corrispondere all'insieme dei punti rappresentante l'area di gioco, ovvero l'area nella quale il giocatore è libero di muoversi, che definisce il rettangolo tale che:

$$x \in [16, 624], \quad y \in [80, 448] \quad (5)$$

Infine, un frutto **non** può essere generato se il generatore cerca di collocarlo sul serpente o sul muro di gioco; o meglio, in tal caso, il codice lo riposiziona finché non trova una posizione facente parte dell'insieme ammissibile di valori.

Dopo che un oggetto è stato mangiato oppure dopo che il timer relativo al frutto scade, il frutto scompare e viene innescata la generazione di un nuovo frutto.

Non appena il programma viene fatto partire, la posizione iniziale del primo frutto generato è posizionata fuori stage in modo che il frutto non vada a trovarsi su qualche voce del menù principale, creando un fastidioso effetto visivo; non appena si entra nello stato di *playing*, l'algoritmo di selezione dell'oggetto comincia la sua esecuzione.

```

process(CLOCK)
    variable counter_x  : integer range 0 to 40; -- 40 * 16 = 640 px
    variable counter_y  : integer range 0 to 30; -- 30 * 16 = 480 px
    variable counter_z  : integer range 1 to 31;

begin
    if CLOCK'event and CLOCK = '1' then
        if counter_x = 39 then counter_x := 1;
        else counter_x := counter_x + 1;
        end if;

        if counter_y = 28 then counter_y := 5;
        else counter_y := counter_y + 1;
        end if;

        if counter_z = 31 then counter_z := 1;
        else counter_z := counter_z + 1;
        end if;

        if counter_z < 10 then
            item_type_new <= 1;
        elsif counter_z >= 10 and counter_z < 20 then
            item_type_new <= 2;
        elsif counter_z >= 20 and counter_z < 30 then
            item_type_new <= 3;
        else item_type_new <= 4;
        end if;

        -- Item spawn
        item_x_new <= to_unsigned(counter_x * 16, 10);
        item_y_new <= to_unsigned(counter_y * 16, 10);
    end if;
end process;

-- Check whether the condition is respected or not
if ITEM_ATE = '1' or ITEM_ON_WALL = '1' or ITEM_ON_SNAKE = '1' then
    state_next <= get_new_item;
end if;

-- Check if the timer is over, in that case the item needs to disappear
if item_type_register >= 2 and TIMER2_UP = '1' then
    state_next <= get_new_item;
end if;

```

5 Le mappe di gioco

5.1 I livelli di gioco

La fase di creazione e la gestione dei livelli di gioco è stata la più creativa di tutte. Affinché un gioco sia ben progettato è essenziale che i livelli riescano in qualche modo a cambiare l'esperienza dell'utente e a renderla più o meno difficile a seconda del livello selezionato. Tendenzialmente la difficoltà cresce linearmente con l'avanzamento dei livelli di gioco.

Uno stage (o *livello*) è un insieme di blocchi sparsi in una mappa di gioco con un certo criterio; il blocco rappresenta l'unità fondamentale di uno stage e, a seconda della topologia del livello, la difficoltà di gioco può essere più o meno alta. I blocchi, per come è strutturata la logica di gioco, non devono essere toccati dal giocatore altrimenti le vite a disposizione diminuiscono innescando un *game over*.

5.1.1 Posizionamento dei blocchi

Il posizionamento dei blocchi è molto semplice; innanzitutto, così come per la generazione del testo, è necessario definire un *block ROM* che definisce l'estetica del componente da posizionare. Dopodiché, è possibile posizionare il blocco lavorando come se ci trovassimo su un piano cartesiano, in quanto è caratterizzato da una coppia di coordinate (x, y) ed una forma quadrata di lunghezza (e larghezza) pari a 16 pixel.

Per il posizionamento abbiamo lavorato sul tool matematico *desmos* per il graphing di funzioni.

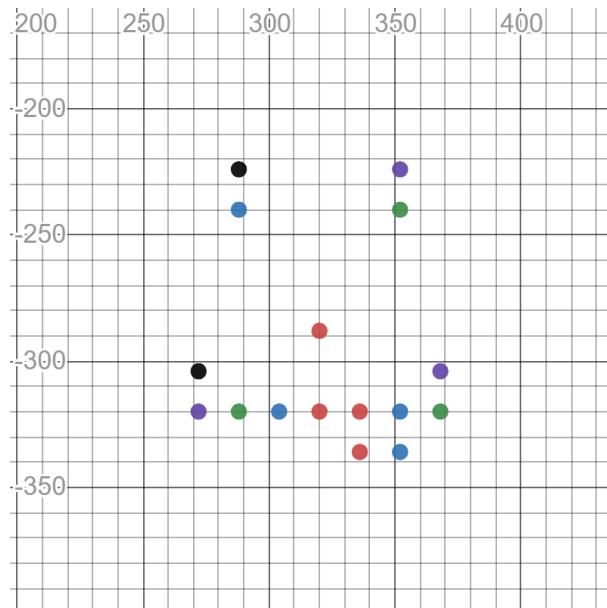


Figura 16: Il tool Desmos

Dev'essere effettuata una trasformazione per utilizzare i livelli disegnati su Desmos: per la precisione, l'operazione prevede un ribaltamento dell'asse delle ascisse. Inoltre, per poter assicurare un corretto posizionamento dei blocchi all'interno dello stage, è necessario seguire specifici vincoli in modo tale da non commettere errori nel disegnare i blocchi e ritrovarli all'esterno dello stage, in punti in cui il giocatore non è in grado di raggiungerli.

Abbiamo:

$$(x, y) \leftrightarrow (-x, y), \quad x \in [16, 624], y \in [80, 464]$$

Dopo aver effettuato la suddetta operazione è possibile utilizzare le coordinate in gioco per il posizionamento.

```
wall_x(0) <= to_unsigned(272,10) when stage_select=1 else
                                to_unsigned(1000,10) when stage_select=2 else
                                to_unsigned(0,10);

wall_y(0) <= to_unsigned(224,10) when stage_select=1 else
                                to_unsigned(1000,10) when stage_select=2 else
                                to_unsigned(0,10);
```

Il file *stages.vhd* è in grado di gestire più livelli; se volessimo cambiare la posizione del blocco dello stage 1, basterebbe modificare le coordinate associate alla variabile *wall_y* *wall_x* quando *stage_select* assume il valore 1.

5.2 Il caricamento di livelli esterni attraverso il ROM Editor

Il ROM Editor scritto in Python permette all'utente di poter caricare nuovi livelli in gioco. Al momento, abbiamo progettato 3 set di livelli da provare, ma è possibile utilizzare Desmos per la progettazione di nuovi livelli.

Si ricordi di rispettare i vincoli di posizionamento del blocco, in modo che essi non si poszionino male sullo stage di gioco.

Una volta scritti i livelli in VHDL, bisogna importarli nella cartella *resources* del ROM Editor e salvarli come *setN.dat*. Dopodiché utilizzare il ROM Editor per applicare il nuovo layout.



Figura 17: Cliccare su *apply* per applicare il layout disegnato

6 Le ROM utilizzate

6.1 Generazione del testo

Una funzione molto importante per un qualsiasi video controller è la generazione del testo su schermo. Molto spesso vengono utilizzati appositi circuiti dedicati per facilitare la generazione di caratteri su schermo.

Per mostrare del testo su un display VGA, bisogna organizzare la *display area* dalla risoluzione di 640×480 in *tiles*; ogni *tile* rappresenta la posizione di un carattere. Ogni carattere sarà di dimensioni 8×16 ($l \times h$) pixel.

6.1.1 Font ROM

Il design di ogni carattere è determinato dal cosiddetto *font ROM* (*v. font_ROM.vhd*); esso contiene il pattern di pixel da mostrare a schermo. Se un elemento della matrice 8×16 contiene un 1, allora il pixel viene mostrato in *foreground*; altrimenti, viene mostrato in *background*.

Il font ROM utilizzato (simile al font IBM usato nei vecchi PC) è stato recuperato da Internet, dettagli alla voce bibliografica [4]. È possibile reperire altri tipi di font, a seconda del proprio gusto, ed utilizzarli nel proprio progetto.

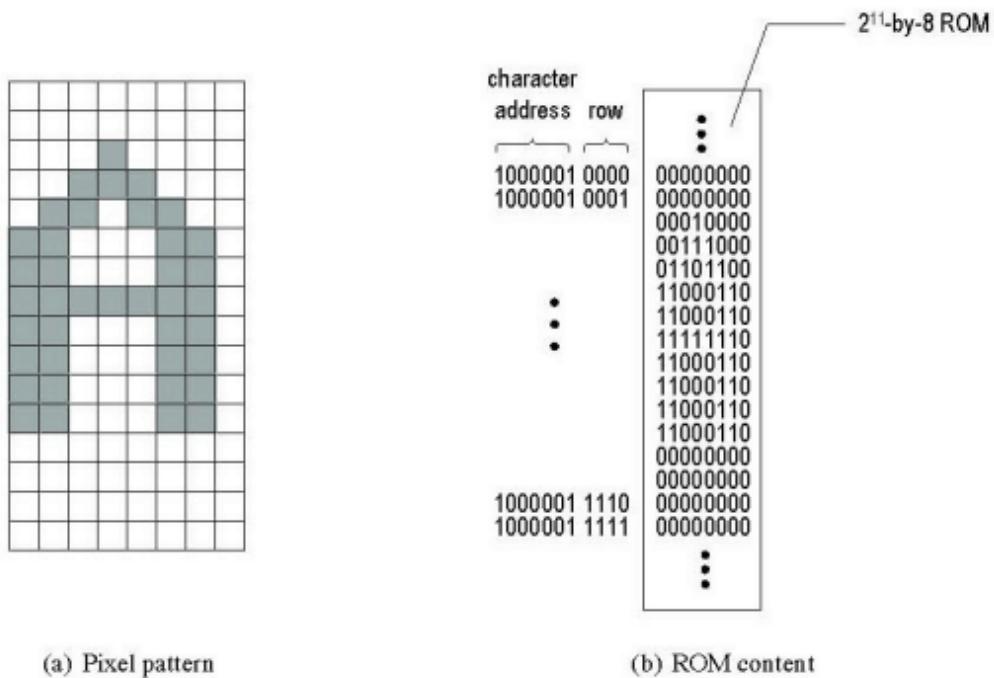


Figura 18: Un esempio di un carattere

```

begin
    process (CLOCK)
        begin
            if (CLOCK'event and CLOCK = '1') then
                address_register <= ADDRESS;
            end if;
        end process;
    DATA <= ROM(to_integer(unsigned(address_register)));
end arch;

```

6.2 Generazione dei bordi

Anche i bordi che definiscono l'area nella quale il serpente è libero di muoversi sono gestiti da un file *ROM* che definisce il loro aspetto. Per il posizionamento sullo stage, l'algoritmo verifica che le condizioni relative alle posizioni siano rispettate, ed, in quel caso, si provvede al disegno dei bordi in quell'area di gioco.

```

-- show border
border_on    <= '1' when (0 < unsigned(PIXEL_X) and unsigned(PIXEL_X) < 3)
              or (638 < unsigned(PIXEL_X) and unsigned(PIXEL_X) < 641)
              or (0 < unsigned(PIXEL_Y) and unsigned(PIXEL_Y) < 3)
              or (478 < unsigned(PIXEL_Y) and unsigned(PIXEL_Y) < 481)
              or (79 < unsigned(PIXEL_Y) and unsigned(PIXEL_Y) < 82)
              else '0';

```

I vincoli numerici presenti nel codice rappresentano il luogo dei punti del piano nei quali i bordi devono essere disegnati.

6.3 Generazione degli altri elementi

Sono stati gestiti nella stessa maniera anche altri elementi di gioco tra i quali la *testa del serpente*, il *corpo del serpente*, la *level picker icon*, tutti i *frutti normali / speciali* ed infine gli *ostacoli di gioco*. Per rendere facile la loro modifica e una personalizzazione del gioco, è possibile modificare tali elementi nel file package del progetto (*v. the_snake_game_package.vhd*).

```
constant WALL_ROM: rom_block := (
    "1111111100011111",
    "1111111100000111",
    "1111111100000001",
    "1111111100000000",
    "1111111100000000",
    "1111111100000001",
    "1111111100000011",
    "1111111100001111",
    "1111111100011111",
    "1111100011111111",
    "1110000011111111",
    "1000000011111111",
    "0000000011111111",
    "0000000011111111",
    "1000000011111111",
    "1110000011111111",
    "1111100011111111" );

constant SNAKE_HEAD_ROM: rom_block := (
    "0000000000000000",
    "000011111110000",
    "000111111111000",
    "001111111111100",
    "0110001111000110",
    "0110001111000110",
    "0110001111000110",
    "011111111111110",
    "011111111111110",
    "011111111111110",
    "011111111111110",
    "011111111111110",
    "001111111111100",
    "000111111111100",
    "0000111111111000",
    "0000000000000000" );
```

6.4 Recupero dei caratteri

Per recuperare un carattere dalla *font ROM* è possibile riferirsi al suo indirizzo. La ROM tratta tutto l'alfabeto come un'unica grande costante, ovvero un unico grande blocco. Estrapoliamo parte dell'alfabeto in uno snippet e mostriamo un semplice esempio:

```
-- code x45
"00000000", -- 0
"00000000", -- 1
"11111110", -- 2 ******
"01100110", -- 3 ** **
"01100010", -- 4 ** *
"01101000", -- 5 ** *
"01111000", -- 6 ****
"01101000", -- 7 ** *
"01100000", -- 8 **
"01100010", -- 9 ** *
"01100110", -- a ** **
"11111110", -- b ******
"00000000", -- c
"00000000", -- d
"00000000", -- e
"00000000", -- f

-- code x46
"00000000", -- 0
"00000000", -- 1
"11111110", -- 2 ******
"01100110", -- 3 ** **
"01100010", -- 4 ** *
"01101000", -- 5 ** *
"01111000", -- 6 ****
"01101000", -- 7 ** *
"01100000", -- 8 **
"01100000", -- 9 **
"01100000", -- a **
"11110000", -- b ****
"00000000", -- c
"00000000", -- d
"00000000", -- e
"00000000", -- f
```

Se volessimo scrivere le iniziali del prof. Eugenio Faldella, ovvero le lettere *E* e *F*, dobbiamo prima riferirci al codice della *E*, ovvero 0×45 e poi alla lettera *F*, ovvero 0×46 .

Quindi il video controller leggerà in una specifica porzione dello schermo e quando gli sarà dato il comando di disegno, la lettera *E*, ed, dopo essersi spostato di una certa quantità di pixel sull'asse delle ascisse, a seconda della spaziatura desiderata, la lettera *F*.

Continuando su questa strada è possibile scrivere intere stringhe, dando vita a GUI esplicative ed utili ai fini di un qualsiasi programma dotato di interfacciamento video.³

**SIS. DIGITALI – PROF. FALDELLA
CESARANO, CROCE**

Figura 19: La firma di gioco

³Se la risoluzione fosse più alta di quella adottata, la matrice necessaria di pixel dovrebbe essere più grande in modo da non avere scritte molto piccole in video.

7 Le animazioni di gioco

7.1 Animazione della GUI

Le animazioni servono a rendere l'esperienza di gioco più dinamica e divertente. Per la realizzazione di un'animazione si utilizzano dei timer che commutano gli stati di un oggetto ripetutamente ad una certa frequenza, sostituendone, per esempio, lo *sprite* di gioco (immagine in grafica raster rappresentante l'oggetto) con uno simile. Il continuo susseguirsi (a frequenze elevate) di questo *replace* di *sprite* crea l'effetto visivo che oggi è conosciuto come animazione.

Per la realizzazione in VHDL, abbiamo implementato un primo timer per la gestione della presenza degli elementi testuali grafici sullo schermo.

Scritte come **Stage Clear** o **Level up** devono scomparire dopo un lasso di tempo predefinito deciso dall'utente. Per la sua realizzazione, si utilizza un timer che conta a ritroso dove lo 0 rappresenta il punto di arrivo, ovvero la fine del conteggio.

Il timer funziona nel seguente modo: si salva il valore del counter in un registro e lo si aumenta con una certa frequenza. Più la frequenza è alta, più il timer conta velocemente.

La velocità di conteggio di un timer è dettata dal Refresh Rate del dispositivo. Nel nostro caso, la frequenza di aggiornamento è di 60 Hz/s ; per contare due secondi come nel nostro caso, abbiamo bisogno di una stringa a 7 bit.

Abbiamo:

$$t = \frac{2^7}{60} = \frac{128}{60} \approx 2.1 \text{ s} \quad (6)$$

Alleghiamo un piccolo snippet per mostrare il funzionamento del timer in VHDL:

```
-- The counters we used count backwards. The logic of a timer is clearly one
architecture arch of timer is

    signal timer_register, timer_next : unsigned(6 downto 0);

begin

    -- registers
    process (CLOCK, RESET)
    begin

        -- RESET State: we need to initialize every value.
        if RESET = '1' then
            timer_register    <= (others =>'1');
        elsif (CLOCK'event and CLOCK = '1') then
            timer_register    <= timer_next; -- present-state logic
        end if;

    end process;

    -- next-state logic
    process(TIMER_START, timer_register, REFRESH_TICK)
    begin

        if (TIMER_START = '1') then
            timer_next <= (others=>'1');

        elsif REFRESH_TICK ='1' and timer_register /= 0 then
            timer_next <= timer_register - 1; -- tick reached, decrease value until 0
        else
            timer_next <= timer_register;
        end if;

    end process;

    -- when we reached 0 we set TIMER_UP to 0. Useful for handling animations.
    TIMER_UP <= '1' when timer_register = 0 else '0';

end arch;
```

7.2 Animazione dei frutti

In maniera molto simile al primo timer, il secondo timer è stato utilizzato per i frutti speciali di gioco. In questo caso, i frutti speciali appaiono per un lasso di tempo pari a:

$$t = \frac{2^9}{60} = \frac{512}{60} \approx 8.52 \text{ s} \quad (7)$$

```
architecture arch of timer2 is

    signal timer2_register, timer2_next: unsigned(9 downto 0);
begin
    process (CLOCK, TIMER2_RESET) -- registers
    begin
        if TIMER2_RESET = '1' then
            timer2_register <= "1000000000";

        elsif (CLOCK'event and CLOCK = '1') then
            timer2_register <= timer2_next;
        end if;
    end process;

    -- next-state logic
    process(TIMER2_START, timer2_register, REFRESH_TICK)
    begin
        if (TIMER2_START = '1') then
            timer2_next <= "1000000000";

        elsif REFRESH_TICK = '1' and timer2_register /= 0 then
            timer2_next <= timer2_register - 1;

        else
            timer2_next <= timer2_register;
        end if;
    end process;

    TIMER2_UP <= '1' when timer2_register = 0 else '0';

end arch;
```

8 Conclusione

Il progetto è riuscito nel suo intento, nel corso di questi mesi siamo riusciti ad apprezzare la potenza e la bellezza della programmazione su FPGA e del linguaggio VHDL, seppur senza alcun tipo di supporto da parte del docente.

Sviluppare per VHDL ci ha migliorato come Ingegneri e, più precisamente, come *ingegneri del software* poiché abbiamo imparato (...a nostre spese!) a tener ben separate le fasi di *Progettazione* e di *Programmazione* del nostro sistema, e di questo ne siamo molto grati.

8.1 Sviluppi futuri

L'intenzione è di lasciare il progetto in formato open-source e pubblicato su *github.com* in modo che possa esser consultato da altri studenti e, con grande probabilità, migliorato. Ci farebbe piacere ricevere un feedback da chiunque abbia intenzione di usare il nostro progetto.

Di seguito elenchiamo possibili migliorie del progetto:

- Implementare supporto all'audio Wolfson WM8731;
- Implementare una leaderboard degli high-score tramite SD-CARD;
- Utilizzare una tavolozza di colori a 4 bit per una paletta di colori più ampia;
- Implementare una *Splash Screen* pre avvio;
- Aggiunta di ulteriori animazioni;
- Blocchi animati che cambiano posizione sullo stage.

8.2 Ringraziamenti

Ci verrebbe da ringraziare il nostro relatore ma, purtroppo, non ci ha fornito supporto in nessuna fase dello sviluppo del progetto. Non consiglio, per questo motivo, di considerare l'inserimento dell'esame di Sistemi Digitali M nel proprio piano di studi.

Il vero ringraziamento va ai tecnici del Lab. 2 che ci hanno sopportato (sì, con la “o”) in questi ultimi mesi fornendoci di continuo le periferiche di supporto per il sistema e supporto con i terminali.

Un grazie anche all'Ing. Pietro Bassi che ha saputo darci feedback continui sul progetto, data la sua esperienza in merito e a Primiano Tucci per l'idea dello script di pulizia della workspace [5].

Infine, un grazie a tutti i nostri colleghi che hanno provato il sistema prima della presentazione aiutandoci a scovare falle, anche piuttosto gravi, di progettazione e programmazione.

Riferimenti bibliografici

- [1] Stefano Suraci. «Linguaggi e ambienti CAD per la sintesi logica di sistemi digitali». Relatore: prof. Eugenio Faldella, Correlatori: Prof. Stefano Mattoccia, Prof. Marco Prandini. Tesi di laurea mag. Alma Master Studiorum - Università di Bologna, 2008-2009.
- [2] Luca Cesarano e Andrea Croce. *Teransic Altera DE1 - The Snake Game in VHDL*. 2019. URL: <https://www.youtube.com/watch?v=dK14xL8vZxY>.
- [3] Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan -3 Version*. 1^a ed. Wiley-Interscience, 2008. ISBN: 0470185317.
- [4] Brigham Young University. *VGA Text Generation*. URL: https://ece320web.groups.et.byu.net/labs/VGATextGeneration/list_ch13_01_font_rom.vhd.
- [5] Primiano Tucci. *Personal Website*. URL: <https://www.primianotucci.com/>.