ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROGETTO FINALE ED ATTIVITÀ PROGETTUALE DI

Sistemi Digitali M

Arkanoid VHDL

Pietro Bassi
Matricola 0000749438

Marco Torsello
Matricola 0000726358

SOMMARIO

Capitolo 1. INTRODUZIONE

1.1 IL GIOCO

Arkanoid è un videogioco Arcade, originariamente sviluppato dalla software house Taito nel 1986. Esso può essere considerato l'erede di Breakout, gioco dalle meccaniche molto simili, distribuito nel 1976 per piattaforme Atari.

L'incredibile popolarità raggiunta da questo titolo ha fatto sì che nel corso degli anni siano state sviluppate decine di differenti versioni, ognuna delle quali caratterizzata da aggiunte e variazioni al gameplay di varia natura, mantenendo comunque immutate le regole e le componenti fondamentali che hanno reso celebre il gioco.

Figura – Il Breakout originale per piattaforma Atari 2600

Il giocatore controlla una navicella (da ora in poi "pad" o "paddle") che ha la funzione di racchetta, in grado di prevenire la caduta di una sfera ("ball") al di sotto del limite inferiore dello schermo, reindirizzandola verso la parte alta dell'area di gioco, all'interno della quale sono presenti dei mattoncini colorati ("brick"). La collisione della pallina con un mattoncino porta alla distruzione di quest'ultimo e al rimbalzo della sfera, in accordo con le leggi della fisica. L'eliminazione di tutti i brick permette di

passare al livello successivo: lo scopo del gioco consiste nel superare tutti i livelli (33, nella versione originale). Qualora la pallina dovesse cadere nella zona sottostante la navicella, il numero di vite ("lives") scenderebbe di uno. L'avventura può terminare precocemente con una sconfitta nel caso in cui il giocatore esaurisse tutte le vite (ripristinate generalmente al valore 3 ad ogni nuova partita). In concomitanza con la distruzione di un mattoncino, con una certa probabilità potrebbe apparire un "power up" che scende lentamente verso il fondo dello schermo: se catturato dalla navicella, esso può garantire vantaggi di vario genere, dall'incremento del numero totale di vite rimanenti al rallentamento della velocità della sfera.

1.2 LA NOSTRA VERSIONE

Nelle fasi preliminari del progetto, in sede di analisi, si è deciso di mantenere immutate le regole fondamentali sopracitate, aggiungendo però alcuni elementi in grado di rendere maggiormente distintiva e piacevole l'esperienza di gioco, fornendo al nostro clone una sua identità.
Nello specifico, queste sono le caratteristiche che si è deciso di implementare:

- Il rimbalzo della sfera sul pad non porta alla banale riflessione della componente verticale della velocità della pallina, bensì, l'angolo di rimbalzo a collisione avvenuta è funzione della posizione relativa della pallina rispetto al punto centrale del pad. Ciò significa che colpire la sfera con la parte centrale del pad porterà ad un rimbalzo quasi verticale, mentre una collisione con le parti più esterne del pad produrrà un angolo in uscita molto più accentuato (una spiegazione dettagliata con un'immagine esplicativa verrà presentata nel capitolo relativo al Datapath). Questa decisione permette al giocatore di direzionare facilmente la pallina, migliorando la giocabilità
- La velocità della sfera aumenta gradualmente con il trascorrere del tempo e viene ripristinata al valore iniziale all'inizio di ogni livello o qualora il giocatore dovesse perdere una vita
- Oltre ai mattoncini "standard" caratterizzati da diverse tonalità, ve ne sono alcuni che richiedono più di una collisione per essere distrutti. Questi brick sono di colore grigio e più scura è la tonalità di questo colore, maggiore è il numero di rimbalzi che la pallina deve effettuare su di essi al fine di eliminarli.
  Vi sono anche i mattoncini oro: essi non possono essere distrutti in alcun modo

e, ovviamente, non vengono calcolati ai fini del completamento del livello; rappresentano tuttavia un elemento che può aumentare la difficoltà di un particolare *stage*

- Alla distruzione di un brick, con una certa probabilità, può apparire un Powerup in grado di portare benefici o svantaggi di diversa natura. I cinque tipi possibili sono **Life**, **Enlarge**, **Slow**, **Fast**, **Death** ed i loro effetti saranno descritti nel dettaglio nell'apposita sezione

Capitolo 2. IL PROGETTO

2.1 PROTOTIPO

Prima di iniziare la progettazione e lo sviluppo in linguaggio VHDL, è stato realizzato in tempi molto ristretti un prototipo del gioco in linguaggio Java. Il pattern *observer* è stato utilizzato per simulare la presenza di un clock periodico e ha fatto sì che, seppur godendo delle potenzialità di un linguaggio di alto livello, questa prima implementazione presentasse concettualmente un logica assimilabile alla struttura del progetto finale su FPGA.

Questa bozza ha permesso di effettuare test iniziali molto rapidi circa i comportamenti di alcuni elementi del gioco (movimento pallina, rimbalzo contro i mattoncini, controllo del paddle, …), che sono stati successivamente implementati nella versione in VHDL e testati attraverso simulazioni basate su Waveform.

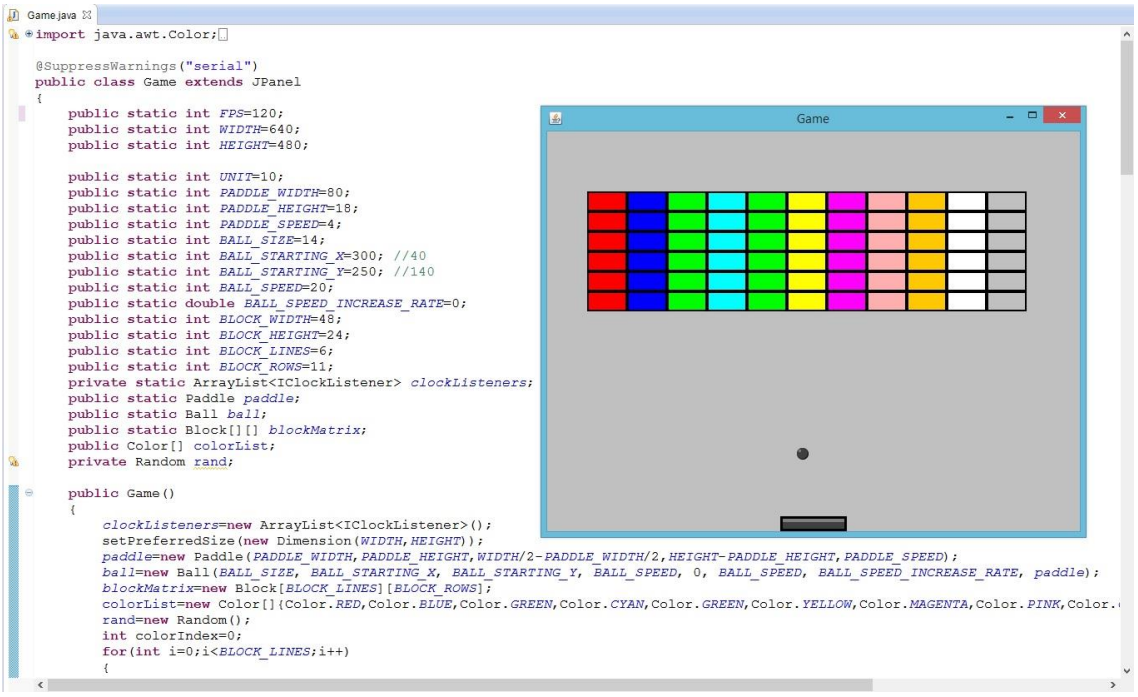


Figura – Prototipo Java

2.2 ARCHITETTURA GENERALE

2.2.1 TOP-LEVEL ENTITY

La top-level entity rappresenta il livello più alto della gerarchia e si interfaccia direttamente con la board DE1. Di quest'ultima, *arkanoid.vhd* sfrutta i seguenti pin:

```
port
(
	--fonte primaria del Clock
	CLOCK_50				: in  std_logic;
	--bottoni fisici DE1 per attivare il segnale di RESET
	KEY					: in  std_logic_vector(3 downto 0);
	--segnali per interfaccia PS/2
	PS2_CLK				: in  std_logic;
	PS2_DAT				: in  std_logic;
	--segnali VGA
	VGA_R					: out std_logic_vector(3 downto 0);
	VGA_G					: out std_logic_vector(3 downto 0);
	VGA_B					: out std_logic_vector(3 downto 0);
	VGA_HS				: out std_logic;
	VGA_VS				: out std_logic;
	--pin per generare suoni tramite buzzer esterno
	GPIO_0				: out std_logic_vector(35 downto 0)
);
end;
```

La struttura interna del progetto segue il pattern *MVC*, implementandolo attraverso la scomposizione in tre moduli principali che interagiscono tra loro, definiti nella top-level entity:

- **Datapath**: gestisce i dati relativi ai componenti del gioco (pallina, paddle, matrice dei mattoncini, powerup) ed è suddiviso in diversi *process*, ognuno dei quali delinea e regola il comportamento dei vari elementi presenti nel livello. Comunica alla *View* le posizioni dei vari oggetti all'interno della scena ed interagisce con il controller al fine di inviare informazioni critiche relative alla logica di gioco e di rispondere correttamente agli eventuali cambiamenti di stato
- **Controller**: contiene e gestisce lo stato principale del gioco ("pausa", "vittoria", "sconfitta", …) e, sulla base di tale informazione, invia a *Datapath* e *View* appropriati segnali. È anche responsabile del controllo degli input dell'utente, che riceve da un modulo adibito all'interpretazione del protocollo utilizzato da una tastiera dotata di interfaccia PS/2

- **View**: trasforma le informazioni che riceve dal *Datapath* e dal *Controller* (rispettivamente posizioni degli oggetti e stato) in elementi visuali e si occupa di mostrarli su un determinato dispositivo di output (VGA). È incaricato anche delle animazioni degli oggetti presenti nella scena

A questi tre moduli principali se ne aggiungono altri di dimensione più limitata, che interagiscono con i precedenti (la loro struttura interna verrà esposta nel dettaglio in seguito):

- **PLL**: viene utilizzato uno dei quattro circuiti phase-locked loop messi a disposizione dalla DE1 per generare il segnale di pixel clock richiesto dallo standard VGA 640x480 @ 60 Hz. Tale frequenza è 25.175 MHz che, approssimata, porta all'utilizzo di un semplice divisore di clock che dimezza la frequenza originale del segnale CLOCK_50, ottenendo un'uscita a 25 MHz (ulteriori dettagli sui timing richiesti dalla VGA saranno esposti nella sezione apposita). I due output del PLL (clock a frequenza originale e dimezzata) sono poi collegati a tutti gli ingressi dei moduli che necessitano tali input
- **Levels ROM**: realizzata sfruttando gli *M4K blocks* presenti nella scheda dimostrativa Altera, essa è costituita da 2048 word da 4 bit ciascuna, per un totale di 8192 bit, nei quali è contenuta la struttura di ognuno dei 13 livelli del gioco
- **Keyboard**: modulo che si interfaccia con la porta PS/2 e permette l'interazione dell'utente tramite una tastiera compatibile con tale protocollo
- **SqrtX** e **SqrtY**: questi due piccoli componenti permettono di calcolare la radice quadrata rispettivamente delle componenti orizzontali e verticali della velocità della pallina
- **Sound**: modulo per la riproduzione di suoni, collegato ad un buzzer esterno tramite il pin GPIO_0(0)

L'entity principale contiene anche il process `game_time_generator`, adibito alla generazione del segnale `gameTime`. Quando quest'ultimo si attiva, per un tempo pari ad un singolo periodo di clock, si ha la valutazione e l'esecuzione di tutte le azione relative alla logica di gioco, dal movimento della pallina a quello del paddle e dei powerup. All'interno del process vi è un contatore che aumenta di uno il suo valore ad ogni fronte positivo del clock, fino al raggiungimento della soglia, data dalla costante

`GAME_LOGIC_UPDATE_RATE - 1`. Tale valore è stato impostato a 416667, garantendo l'aggiornamento della game-logic circa 120 volte al secondo, per un'esperienza di gioco sufficientemente fluida.

2.2.2 PACKAGES

Nel progetto sono stati utilizzati due package, contenenti costanti e funzioni di utilità:

- *Arkanoid_package.vhd*: contiene costanti relative al gioco, alla pallina, al paddle, ai mattoncini e ai powerup, oltre alla definizione dei tipi di dato condivisi dai diversi moduli e funzioni per la valutazione delle collisioni tra pallina e mattoncini
- *Vga_package.vhd*: definisce i valori necessari per il protocollo di comunicazione con la VGA e funzioni e costanti che rappresentano i colori degli elementi nella scena

2.3 SPAZIO DI GIOCO E SPAZIO VGA

Fin da subito si è deciso di adottare come risoluzione lo standard industriale per la VGA 640x480 (@ 60 Hz). Le dimensioni di tale griglia hanno però suscitato alcune perplessità in fase di analisi. Come esposto precedentemente nel capitolo introduttivo, vi era la volontà di ottenere un buon controllo da parte del giocatore per quanto concerne il rimbalzo della pallina sul paddle. Inoltre, si voleva garantire un certo grado di realismo al movimento della sfera, assicurandosi dunque che la somma dei quadrati delle componenti orizzontali e verticali della velocità della pallina rimanesse sempre costante (nonostante le variazioni di direzione della sfera) e che l'aumento di velocità nel tempo fosse graduale. Una griglia di tali dimensioni è apparsa dunque troppo grossolana: aggiornando la game-logic (e quindi la posizione della pallina) 120 volte al secondo, il range di valori interi da attribuire alla velocità sarebbe stato estremamente limitato, non permettendo il rispetto di nessuna delle feature sopracitate.
Proprio per questo motivo, si è deciso di separare lo spazio di gioco "reale", sulla base del quale vengono effettuati tutti i calcoli e relativamente al quale vengono memorizzati i valori di posizione e velocità, da quello della VGA, adibito esclusivamente alla visualizzazione finale degli oggetti.

Lo spazio di gioco rappresenta dunque un sistema di coordinate notevolmente più grande di quello originale e precisamente è dato da una griglia 640*UNIT x 480*UNIT, dove UNIT è una costante impostata a 16 (spazio risultante: 10240x7680).
Il numero 16 non è scelto in maniera casuale: essendo una potenza di due, la conversione di un valore dallo spazio di gioco allo spazio VGA è estremamente efficiente e richiede un quantitativo minimo di hardware aggiuntivo (arithmetic right shift di 4 bit). Tale trasformazione avviene solo all'ingresso del modulo *View*, al quale i dati giungono già pronti per essere rappresentati nella griglia del display 640x480.
Test empirici, inizialmente sul prototipo Java e poi sull'implementazione VHDL, hanno dimostrato come questo approccio, basato su una necessaria approssimazione in sede di visualizzazione su un display output, risulti perfettamente fluido e non presenti nessun comportamento grafico anomalo o sgradevole.

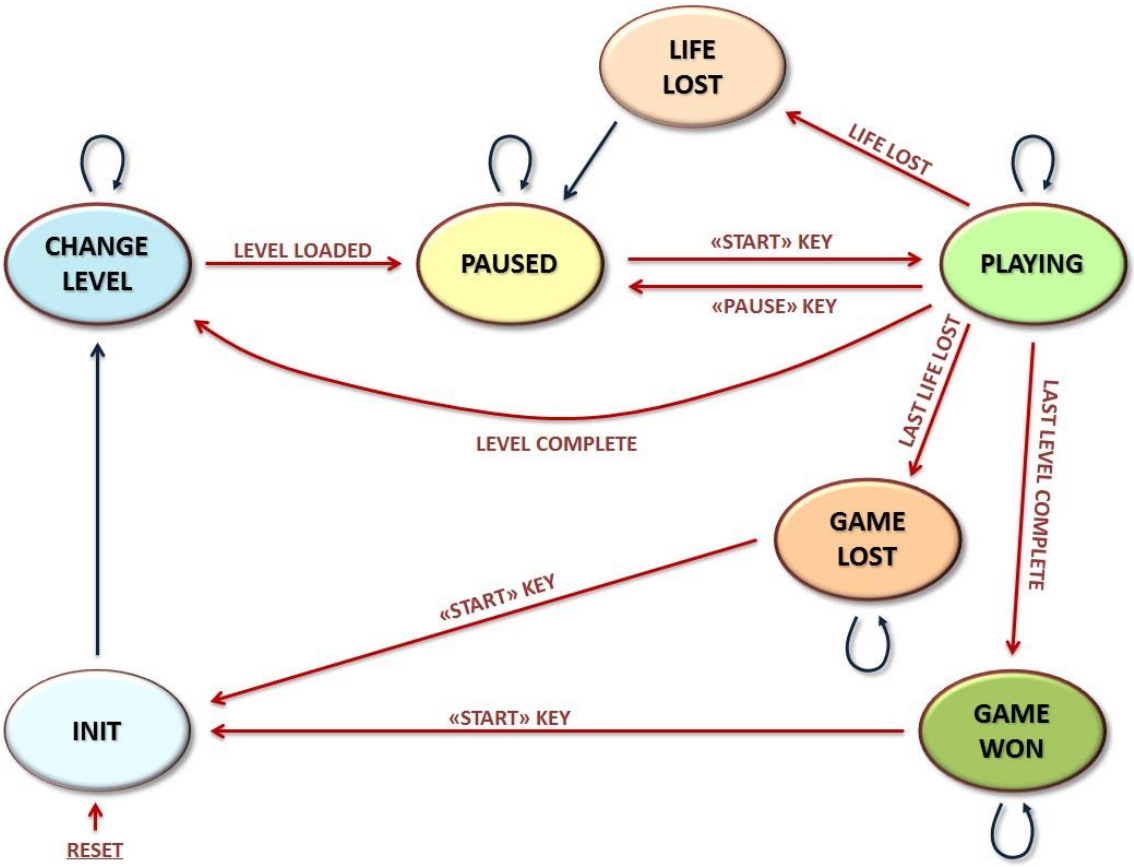
Capitolo 3. CONTROLLER

3.1 STATO

Il *Controller* contiene lo stato attuale del gioco e si occupa del suo cambiamento. Gli stati possibili sono sette e sono elencati in un tipo enumerativo:

```
type state_type is (S_INIT, S_PAUSED, S_PLAYING, S_CHANGELEVEL,
S_LIFELOST, S_GAMELOST, S_GAMEWON);
```

La loro gestione avviene attraverso una macchina a stati di Moore, descritta all'interno di un apposito *process*. Il *Controller* riceve i segnali di input dagli altri moduli e provvede eventualmente a modificare lo stato; quest'ultimo sarà poi propagato a *Datapath* e *View* affinché tali entity possano agire correttamente sulla base di questa informazione. Nella seguente immagine è possibile osservare la *state machine*, nella quale le frecce rappresentano un cambiamento dovuto al verificarsi di fronte positivo del clock. Le frecce rosse, corredate di un condizione necessaria posta sopra di esse, indicano la necessità che un determinato segnale sia attivo al momento del fronte positivo del clock affinché sussista il cambiamento di stato.



- **INIT**: stato iniziale, innescato all'accensione e al reset. Come è deducibile dall'immagine, si tratta di uno stato "temporaneo": esso infatti persiste solo per un periodo di clock, commutando autonomamente e senza necessità di stimoli aggiuntivi al fronte successivo.

  Durante INIT, i segnali di gioco relativi alla partita in corso (come il numero di vite rimanenti) vengono ripristinati ai valori di default e il registro contenente l'indirizzo per l'accesso ai dati della rom dei livelli viene posto a zero. Insieme a quest'ultimo, vengono azzerati anche i segnali di livello corrente, quello relativo al numero di mattoncini da distruggere per passare allo stage successivo e il segnale LEVEL_LOADED che, quando attivo, indica l'avvenuto completamento del caricamento dei dati del livello dalla rom.

  Relativamente alla *View*, INIT causa il reset di tutti i segnali afferenti il protocollo per la VGA
- **CHANGE LEVEL**: vengono ripristinati tutti i valori inerenti la pallina (velocità e posizione iniziale), il paddle (posizione e dimensioni) ed eventuali powerup in caduta scompaiono. Durante questa fase avviene il caricamento di un nuovo livello, descritto in dettaglio nella sezione relativa al *Datapath*. Una volta completate le letture di word dalla rom, il segnale LEVEL_LOADED proveniente dal *Datapath* provoca il passaggio allo stato PAUSED
- **PAUSED**: questo stato, che può essere invocato arbitrariamente dall'utente, "congela" istantaneamente ogni elemento del livello. Esso è anche lo stato in cui si trova il gioco all'avvio e ad ogni cambio di livello, affinché la pallina non inizi a muoversi quando il giocatore è ancora impreparato, causando l'immeritata perdita di una vita. Durante questo stato il gioco è insensibile ad ogni tipo di input, eccezion fatta per la pressione della barra spaziatrice (evento "<<START>> KEY"), la quale fa riprendere l'azione di gioco
- **PLAYING**: ogni elemento di gioco esegue la sua routine ordinaria e l'utente può gestire la posizione del paddle con i tasti 'Y' ed 'I', controllandone il movimento laterale, oltre ad invocare lo stato PAUSED, tramite la pressione del tasto 'P' sulla tastiera. Durante la fase di gioco effettiva, il *Datapath* può inviare al *Controller* due segnali, che possono portare a due stati differenti:
  - *LIFE_LOST*: attivo quando la pallina cade oltre il limite inferiore dello schermo. Se il giocatore è rimasto senza vite, questa perdita porta alla

sconfitta definitiva e lo stato commuta in “GAME LOST”. Se il giocatore ha a disposizione altre vite, si passa nello stato temporaneo “LIFE LOST”.

- *LEVEL_COMPLETE*: attivo quando viene distrutto l’ultimo dei mattoncini distruttibili del livello corrente. Se tale livello era l’ultimo (il tredicesimo), il gioco è stato completato, il giocatore ha vinto e si passa nello stato “GAME WON”. Se non si era ancora arrivati allo stage finale invece, viene caricato il livello successivo, passando per “CHANGE LEVEL”

- **LIFE LOST**: come “INIT”, questo stato persiste solamente per un ciclo di clock, durante il quale i segnali relativi agli elementi della scena vengono ripristinati ai valori di default, come accade nel caso di “CHANGE LEVEL”, con l’unica differenza che, in questo caso, non si passa al livello successivo. “LIFE LOST” commuta automaticamente nello stato “PAUSED”
- **GAME LOST** e **GAME WON**: la partita è terminata e la *View* deve rappresentare graficamente l’esito finale della sessione di gioco (nello specifico, modificando il colore dello sfondo in verde o rosso, rispettivamente in caso di vittoria o sconfitta). Indipendentemente dal fatto che il giocatore abbia raggiunto e completato l’ultimo livello oppure abbia terminato le vite prima della condizione di vittoria, il gioco deve ricominciare dall’inizio: si transita dunque di nuovo nello stato “INIT” tramite la pressione della barra spaziatrice, che causa l’attivazione del segnale “<<START>> KEY”.

  È bene però notare come, alla pressione di tale tasto, lo stato commuti, senza necessità di ulteriori input, in “INIT”, poi in “CHANGE LEVEL” ed infine in “PAUSED”. Tutto ciò avviene in meno di 200 cicli di clock e, considerando che tale segnale si ripete con frequenza 50 MHz, è lecito pensare che, per quanto siano veloci i riflessi del giocatore, la barra spaziatrice risulti ancora premuta. Ciò fa sì che “<<START>> KEY” sia ancora attivo quando ci si trova nello stato “PAUSED”, causando l’inizio istantaneo della nuova partita, senza concedere all’utente il tempo di predisporsi mentalmente alla nuova sessione. Per questo motivo è stato inserito un segnale chiamato *startRegister* che, ad ogni clock, subisce l’assegnamento `startRegister<=BUTTON_START;`.

  Discriminando l’avvenuta pressione del tasto (ed il conseguente cambiamento di stato) con lo statement `”if (BUTTON_START='1' and startRegister='0') then --change state ...”` , si ha che la commutazione di stato avviene solo se la barra spaziatrice era stata precedentemente rilasciata (Monoimpulsore A).

3.2 INPUT UTENTE

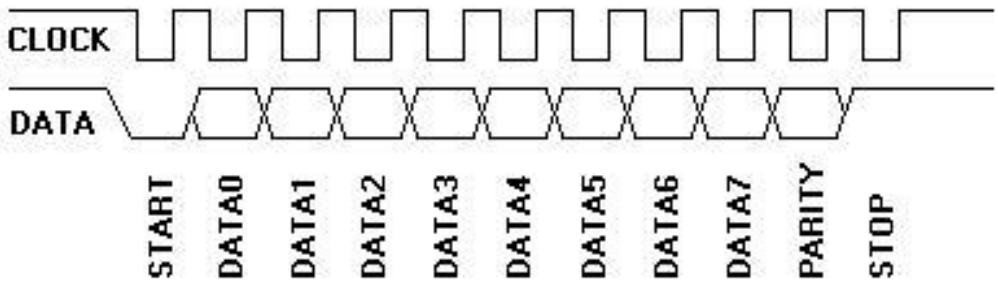
3.2.1 SEGNALI

Il *Controller* si occupa anche di ricevere gli input dell'utente, ricevendo segnali dal modulo *arkanoid_keyboard*, adibito alla gestione di una tastiera PS/2, propagandoli poi al *Datapath* che effettuerà le opportune modifiche ai dati.
I segnali possibili sono quattro:

- KEY_PAUSE: mette in pausa il gioco
- KEY_START: fa riprendere l'azione dallo stato di pausa
- KEY_LEFT: muove il paddle verso sinsitra
- KEY_RIGHT: muove il paddle verso destra

3.2.2 PROTOCOLLO PS/2

Il protocollo utilizzato dalla tastiera viene gestito all'interno di *arkanoid_keyboard*. Il modulo riceve in input due segnali: KEYBOARD_CLOCK (clock generato dal dispositivo) e KEYBOARD_DATA, rappresentante i dati inviati dalla tastiera. Nello specifico, questi ultimi possono essere concettualmente suddivisi in *frame* di lunghezza 11 bit.



Precisamente, si avrà:

- 1 bit iniziale: sempre 0
- 8 bit di dato: dal più significativo al meno significativo
- 1 bit: per la parità

- 1 bit di stop: sempre 1

Alla pressione di ogni tasto corrisponde un codice di *make*, costituito da 8 bit, comunemente espresso da due simboli esadecimali. Ad alcuni comandi (frecce direzionali, Insert, Home, Delete, …) corrispondono codici di *make* più lunghi.
Nel nostro caso:

- KEY_PAUSE: tasto 'P', make code "4D"
- KEY_START: tasto 'SPACE', make code "29"
- KEY_LEFT: tasto 'Y', make code "35"
- KEY_RIGHT: tasto 'I', make code "43"

Per indicare il rilascio di un determinato comando, viene inviato un codice di *break*, costituito da due frame successivi (come per *make* vi sono casi particolari, che non verranno però trattati): nel primo, gli 8 bit di dato rappresentano il numero esadecimale "F0", mentre il secondo frame è uguale al codice di *make* del tasto appena rilasciato.

Il codice segue dunque il protocollo PS/2, salvando il byte di dati nel segnale "code", discriminando il tipo di tasto premuto in un *case-statement* ed inviando in seguito gli opportuni segnali al *Datapath*.
Un *flag* viene attivato quando il codice è "F0", cosicché il modulo sappia che il codice successivo rappresenterà il rilascio del tasto corrispondente, con conseguente abbassamento dei segnali diretti al *Datapath*.

Capitolo 4. DATAPATH

4.1 STRUTTURA

Il *Datapath* è uno dei moduli più complessi dell'intero progetto e per questo motivo la sua organizzazione è stata suddivisa in molteplici process che interagiscono tra loro tramite l'uso di appositi segnali. Nello specifico, si è scelto di creare un process per ognuno degli elementi principali del gioco, ottenendo dunque:

- **Ball process**: gestisce la posizione e il movimento della pallina, effettuando il check di eventuali collisioni ed aggiornando il numero di vite qualora la sfera dovesse superare il limite inferiore; è concettualmente legato al process `SpeedProcess`, adibito all'aumento graduale della velocità della pallina
- **Paddle process**: permette il movimento del pad sulla base dell'input che proviene dal *Controller*
- **Powerup process**: si occupa di effettuare una valutazione probabilistica ogniqualvolta un mattoncino viene distrutto, al fine di determinare se è opportuno o meno far comparire un powerup. Una volta presa questa decisione, il process è incaricato del movimento discendente del powerup e della valutazione relativa all'attivazione di esso, nel caso venga catturato dal paddle
- **Level process**: ha lo scopo di caricare in un primo momento il livello dalla ROM; successivamente, durante lo stato di "PLAYING", si occupa dell'aggiornamento della matrice di mattoncini sulla base di eventuali collisioni avvenute con la pallina e tiene traccia del numero di mattoncini ancora da eliminare per passare al livello successivo

Ognuno dei process elencati contiene nella propria sensitivity list il segnale di Clock e tutto il codice all'interno è subordinato alla veridicità dello statement condizionale `"if(rising_edge(CLOCK)) then"`, che garantisce la sincronicità di tutta l'architettura.

Inoltre, all'interno di ogni process, l'esecuzione dei vari blocchi di codice è discriminata sulla base dello stato attuale (PLAYING, PAUSED, LIFELOST, …), permettendo una chiara divisione delle istruzioni che devono essere eseguite nelle varie fasi del gioco (esempio per LevelProcess: reset dell'indirizzo della rom in "INIT", caricamento del livello in "CHANGELEVEL", controllo ed eventuale aggiornamento della matrice dei

mattoncini in “PLAYING”).

All’interno dello statement condizionale relativo allo stato “PLAYING” vi è anche `”if(GAME_LOGIC_UPDATE='1') then”`, contenente la sezione di codice che viene valutata solo in concomitanza con l’attivazione dell’apposito segnale di update della game-logic ad opera della top-level entity.

4. 2 BRICKS

4.2.1 RAPPRESENTAZIONE

I mattoncini sono blocchi di 40x20 pixel, organizzati all’interno di una matrice 10x15, realizzata tramite l’apposito tipo di dato `type brick_matrix_type is array(natural range <>, natural range <>) of brick_type;`, dove `brick_type` è un `subtype`, definito come `unsigned(0 to 3);` . Ogni mattoncino è dunque rappresentato da 4 bit, per un totale di 16 possibili valori:

```
constant B_EMPTY          : brick_type := X"0";
constant B_WHITE          : brick_type := X"1";
constant B_ORANGE         : brick_type := X"2";
constant B_CYAN           : brick_type := X"3";
constant B_GREEN          : brick_type := X"4";
constant B_RED            : brick_type := X"5";
constant B_BLUE           : brick_type := X"6";
constant B_PINK           : brick_type := X"7";
constant B_PURPLE         : brick_type := X"8";
constant B_YELLOW         : brick_type := X"9";
constant B_GREY1          : brick_type := X"A";
constant B_GREY2          : brick_type := X"B";
constant B_GREY3          : brick_type := X"C";
constant B_GREY4          : brick_type := X"D";
constant B_GREY5          : brick_type := X"E";
constant B_GOLD           : brick_type := X"F";
```

B_EMPTY indica la mancata presenza di un mattoncino in una determinata cella della griglia, le costanti da B_WHITE a B_GREY1 sono i brick normali (un unico rimbalzo della pallina per essere distrutti), i brick da B_GREY2 a B_GREY5 richiedono di essere colpiti più volte, mentre B_GOLD identifica un mattoncino indistruttibile.

Queste strutture dati fanno sì che ogni livello sia rappresentato da un totale di 600 bit (10 x 15 x 4).

4.2.2 ROM DEI LIVELLI

Come precedentemente accennato, la ROM che contiene i livelli è realizzata sfruttando gli *M4K blocks* della board; con 11 bit di indirizzo e 4 bit di uscita, essa permette, ad ogni ciclo di clock, di prelevare una delle 2048 word da 4 bit contenute al suo interno. I 13 *stage* del gioco sono separati tra loro da 8 word poste a "0": si nota dunque come la memoria a disposizione venga utilizzata quasi nella sua interezza ((15 x 10) x 13 + 12 x 8 = 2046 word, con solo 2 word inutilizzate).

4.2.3 CARICAMENTO DEL LIVELLO

Il caricamento del livello avviene all'interno di *LevelProcess*, durante lo stato di CHANGELEVEL. L'algoritmo, qui riportato a meno di qualche segnale di controllo, è il seguente:

```
if(LEVEL_LOADED='0') then
      ROM_ADDR<=std_logic_vector(romAddr(ROM_ADDR'range));
      romAddr<=romAddr+"00000000001";
      if(i_rom/=BRICK_MAX_ROW) then
            brickMatrix(i_rom,k_rom) <= unsigned(ROM_Q);
            if(unsigned(ROM_Q)>=B_WHITE and unsigned(ROM_Q)<=B_GREY5)then
                  bricksForNextLevel<=bricksForNextLevel+1;
            end if;
            if(k_rom/=BRICK_MAX_COL-1)then
                  k_rom<=k_rom+1;
            else
                  k_rom<=0;
                  i_rom<=i_rom+1;
            end if;
      else
            LEVEL_LOADED<='1';
      end if;
else
      i_rom<=0;
      k_rom<=0;
      LEVEL_LOADED<='0';
      LEVEL_COMPLETE<='0';
end if;
```

Finché il segnale LEVEL_LOADED, che indica l'avvenuto caricamento del livello e che viene inoltrato al controller per il cambiamento di stato da CHANGELEVEL a PAUSED, non è attivo, ad ogni clock il processo inserisce la word attualmente in uscita da ROM_Q in una delle celle di brickMatrix ed incrementa l'indirizzo romAddr che

viene inviato alla ROM. Durante questa fase viene anche valutato il tipo di brick inserito: se esso è distruttibile (ogni brick eccetto B_EMPTY e B_GOLD), viene incrementato di 1 il numero di mattoncini che devono essere distrutti per passare al prossimo livello.
Questo ciclo si ripete finché non si supera l'ultima cella della matrice (`i_rom/=BRICK_MAX_ROW`): a quel punto, LEVEL_LOADED viene attivato per un singolo ciclo di clock, innescando il cambio di stato.

4.2.4 EDITOR DI LIVELLI

Per permettere la veloce creazione e personalizzazione dei livelli del gioco, chiaramente poco intuitiva qualora si dovesse modificare manualmente il file *.mif* di inizializzazione della ROM, è stato realizzato in linguaggio Java un *editor*, tramite il quale è possibile modificare in maniera rapida ed efficace i 13 stage differenti.

Figura – Una schermata dell'editor dei livelli

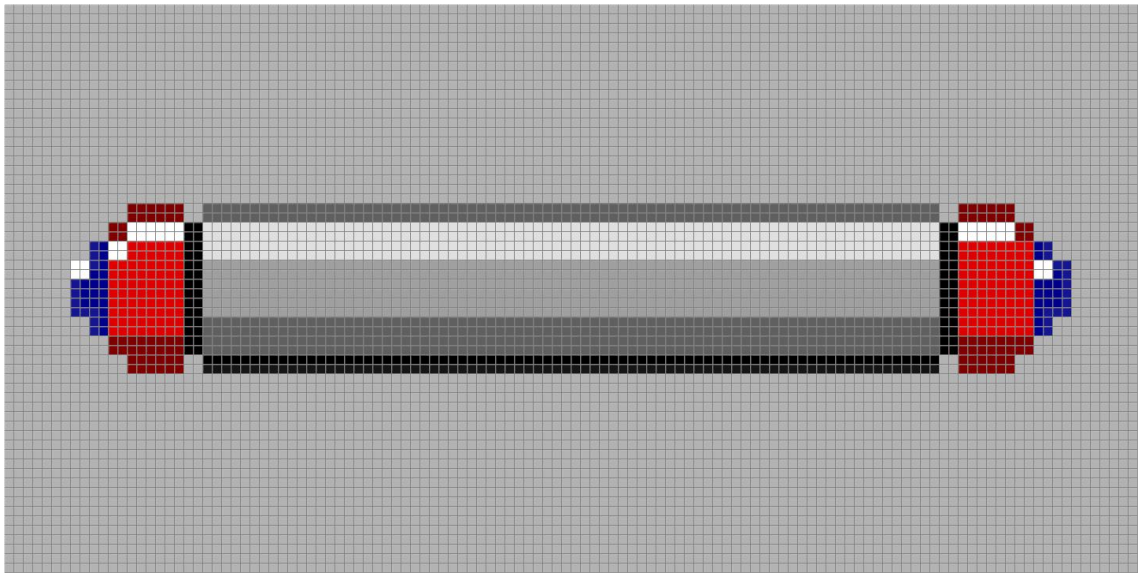
Con un click del mouse si può scegliere il tipo di brick da inserire all'interno della scena tramite la *palette*. I bottoni < e > permettono di passare da uno dei 13 livelli all'altro. Quando si è completata la realizzazione, "Save ROM" esporta la propria creazione in formato *.mif*; sarà sufficiente dunque ricompilare il progetto affinché le modifiche ai livelli abbiano luogo. "Open ROM" carica un file *.mif*, qualora si volesse effettuare una

modifica ad un file di inizializzazione già esistente o si volesse riprendere la creazione di un livello lasciata in sospeso.

4.3 PADDLE

4.3.1 RAPPRESENTAZIONE

Il paddle è rappresentato da un rettangolo 104x18 nello spazio di gioco. È costituito da una parte centrale di colore grigio avente lunghezza 80 e da due parti laterali spesse 12 ognuna. Questa suddivisione non ha valenza esclusivamente grafica ma, come verrà spiegato fra poco, influirà anche sull'angolo di rimbalzo della pallina. Le estremità di colore blu sono luci, caratterizzate da un'animazione lampeggiante.



4.3.2 MOVIMENTO

Il codice relativo al movimento del paddle è qui riportato:

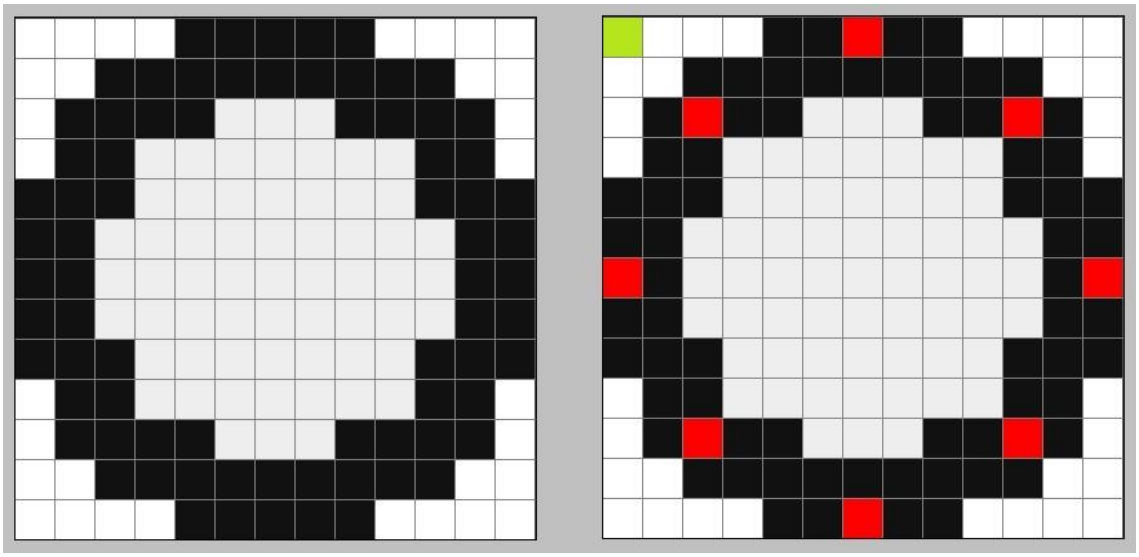
```
--PLAYING
if(STATE=S_PLAYING) then
      nextPaddleX<=paddleX+PADDLE_SPEED_X*PADDLE_MOVE_DIR;
      --game logic update
      if(GAME_LOGIC_UPDATE='1') then
      --paddle movement
            if (nextPaddleX <BOUND_LEFT) then
                  paddleX<=BOUND_LEFT;
            elsif (nextPaddleX+paddleWidth>BOUND_RIGHT) then
                  paddleX<=BOUND_RIGHT-paddleWidth;
            else
                  paddleX<=nextPaddleX;
            end if;
      end if;
end if;
```

Durante la fase di PLAYING, al fronte positivo del clock, viene immagazzinato il valore `paddleX+PADDLE_SPEED_X*PADDLE_MOVE_DIR` nel segnale nextPaddleX. Esso rappresenta la predizione della posizione che il paddle assumerà nella prossima fase di game-logic ed è dato dalla somma della posizione corrente del pad (paddleX), con la velocità PADDLE_SPEED_X (costante) moltiplicata per il segnale PADDLE_MOVE_DIR proveniente dal Controller (0 in mancanza di input, +1 o -1 se l'utente vuole direzionare il pad rispettivamente verso destra o verso sinistra). Qualora essa risulti un posizione valida (il pad non interseca le pareti), nextPaddleX sostituisce la precedentemente posizione, altrimenti paddleX viene arbitrariamente posta alla posizione limite destra o sinistra che il pad può assumere, sulla base dei bordi dell'area di gioco.

4.4 BALL

4.4.1 RAPPRESENTAZIONE

La pallina è rappresentata come un cerchio inscritto in un quadrato di lato 13 pixel. Al fine di poter discriminare correttamente il rimbalzo contro gli altri elementi della scena, sono stati individuati 8 punti lungo la circonferenza, ognuno di questi descritto dalle sue coordinate X ed Y, che approssimano i punti di collisione della sfera (mostrati in rosso nella figura).



Oltre ai suddetti punti, l'oggetto "ball" è descritto dai seguenti segnali:

- **ballX** e **ballY**: coordinate attuali della pallina, che individuano il punto più in alto a sinistra del quadrato circoscritto (pixel verde nell'immagine)
- **X** e **Y**: le coordinate "predette" per il prossimo update della game-logic, qualora la pallina non entri in collisione con nessun oggetto. Sono utilizzate per il controllo degli eventuali rimbalzi
- **squaredSpeed**: il quadrato della velocità attuale, diviso UNIT (il motivo di tale operazione sarà presto spiegato)
- **ballAngleX** e **ballAngleY**: sanciscono in che percentuale squaredSpeed deve essere ripartita nella componente orizzontale e in quella verticale, decretando in sostanza l'angolazione del vettore velocità
- **signX** e **signY**: segni, positivi o negativi, di ballAngleX e ballAngleY, che indicano la direzione delle due componenti. La decisione di separare i segni dai valori permette di collegare direttamente i quadrati delle componenti ai moduli che effettuano il calcolo della radice quadrata, senza necessità di moltiplicare per -1 eventuali numeri negativi

4.4.2 MOVIMENTO

Il concetto alla base del movimento della pallina è il seguente: all'inizio di ogni fase di aggiornamento della game-logic, **ballX** e **ballY** contengono le coordinate correnti della sfera, mentre **X** ed **Y** rappresentano la predizione della posizione sulla base della velocità corrente e sono dunque esprimibili come X=ballX+ballSpeedX e Y=ballY+ballSpeedY.
Si possono ora presentare due casi:

Caso 1): la posizione predetta della sfera è tale che nessuno dei suoi 8 punti di collisione è sovrapposto ad un elemento della scena (paddle, mattoncino, bordi).
ballX e **ballY** vengono quindi posti uguali rispettivamente ad **X** e ad **Y**, poiché le posizioni predette sono fisicamente possibili.

Caso 2): la posizione predetta della sfera interseca un elemento del gioco. In questo caso, la sfera occupa una posizione non realistica e l'assegnamento del *Caso 1* non è

possibile.

A **ballX** e **ballY** saranno dunque assegnati due valori che sono **funzioni di X e Y**, ottenuti tramite simulazione del movimento della pallina (considerando anche il cambio di direzione dovuto alla collisione) durante il lasso di tempo tra una fase di game-logic e la seguente.

Analizziamo ora più in dettaglio questo secondo caso.

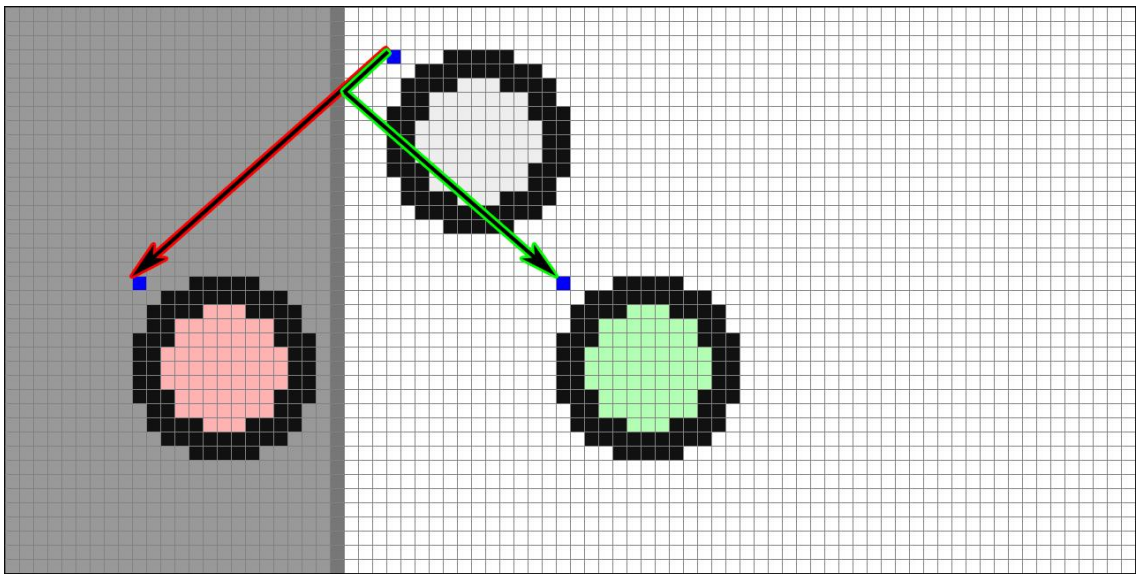
4.5 COLLISIONE CON I BORDI

Si consideri l'esempio più semplice e cioè la collisione con i bordi (sinistro, destro e superiore). Analizziamo il caso della collisione con il bordo sinistro.

Ricordando che *x* aumenta verso destra ed *y* aumenta verso il basso, supponiamo che la velocità della pallina sia descritta dal vettore bidimensionale (-16,18). Questo significa che, ad ogni aggiornamento della game-logic, verranno fatti i seguenti assegnamenti:

- X = ballX - 16
- Y = ballY + 18

Nella figura sottostante la pallina bianca rappresenta la posizione attuale mentre la pallina rossa quella predetta, sulla base del vettore velocità. I pixel blu indicano i punti rappresentativi delle coordinate delle sfere.



Risulta ovvio constatare come la posizione della sfera rossa non sia plausibile dal punto di vista fisico. Si vuole inoltre che tale posizione non venga propagata alla *View*, bensì utilizzata solo come "posizione temporanea" per il calcolo delle coordinate finali corrette, relative alla pallina verde. Quest'ultima rappresenta dunque la posizione che ci si aspetterebbe venisse visualizzata nel prossimo update della game-logic.
Quali sono le sue coordinate?

Il problema è riconducibile all'individuazione di un punto $P2(x_2, y_2)$ simmetrico ad un punto $P1(x_1, y_1)$ rispetto ad una retta x_0 parallela all'asse y.

Relativamente a questo caso specifico, $P2(x_2, y_2)$ è la posizione (pixel blu) della sfera verde, $P1(x_1, y_1)$ è la posizione della sfera rossa e x_0 è l'ascissa del bordo sinistro (linea verticale grigio scuro).

Sulla base di queste ipotesi, è facile ricavare le formule della simmetria, che portano ai seguenti assegnamenti:

- $y_2 = y_1$
- $x_2 = 2\,x_0 - x_1$

In codice VHDL, questo si traduce in:

```
if (ballLeftX<=BOUND_LEFT) then
      ballX<=2*BOUND_LEFT-X;
      signX<=1;
end if;
```

dove ballLeftX è l'ascissa del margine sinistro della pallina mentre BOUND_LEFT rappresenta x_0.
Si noti come, giustamente, il segno della componente orizzontale della velocità venga reso positivo a causa dell'impatto contro il bordo laterale, modificando il vettore velocità in (16,18).

Un ragionamento analogo può essere fatto per la collisione con il bordo superiore, invertendo x ed y.

Nel caso di collisione con il bordo destro invece, è bene notare come non sia più possibile applicare la medesima formula, dal momento che bisogna tenere conto anche delle dimensioni della pallina: questo si concretizza nello "spostamento" della retta di

simmetria x_0 verso sinistra di una distanza pari alla costante BALL_SIZE. Il codice che ne deriva è il seguente:

```
if (ballRightX>BOUND_RIGHT) then
      ballX<=2*BOUND_RIGHT-2*BALL_SIZE-X;
      signX<=-1;
end if;
```

4.6 COLLISIONE CON I MATTONCINI

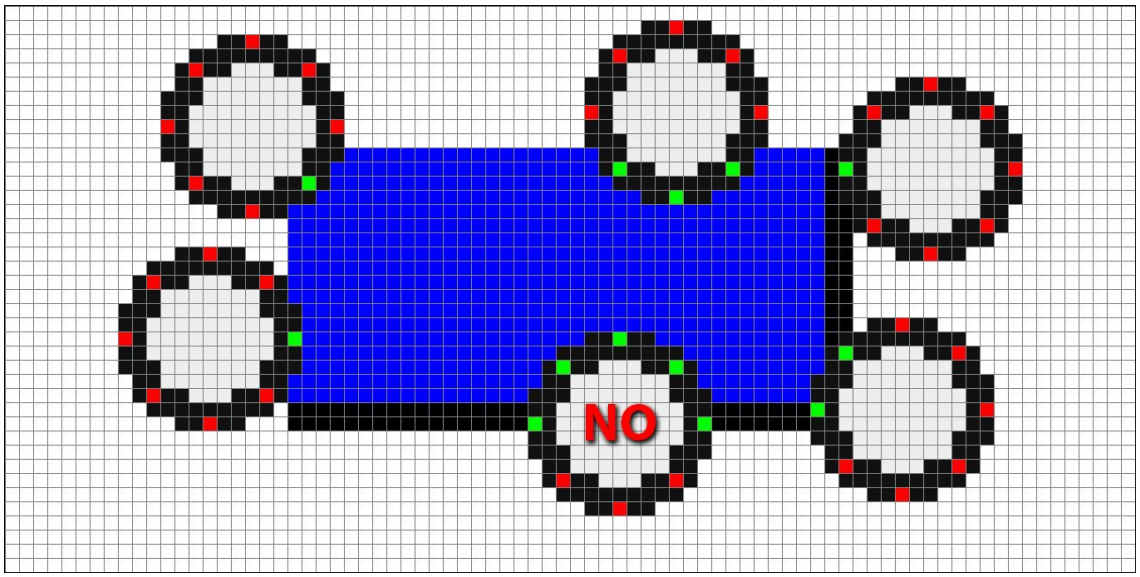
4.6.1 STRATEGIA UTILIZZATA

La tecnica appena descritta è la medesima che viene utilizzata per i mattoncini, con l'unica differenza che, in questo caso, vengono considerati tutti e 8 i punti di bordo della sfera.

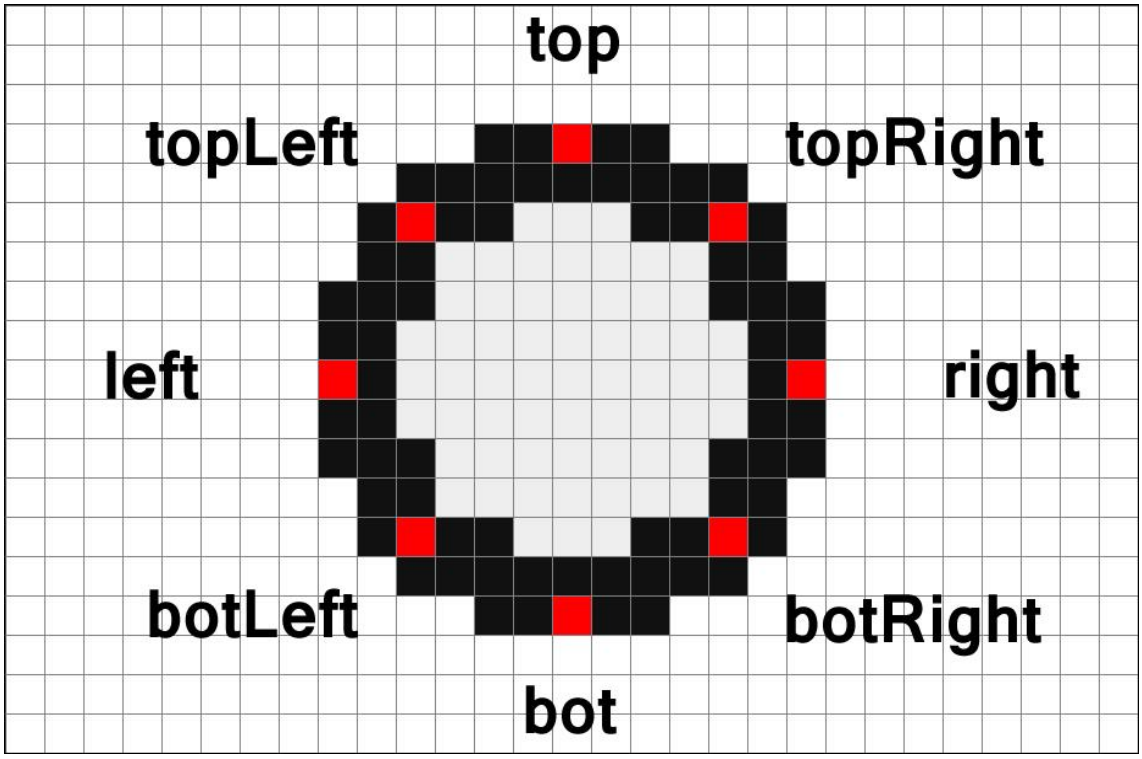
Per valutare l'avvenuta collisione con un mattoncino, viene sfruttata una funzione contenuta in *arkanoid_package.vhd*.
Siano (pointX,pointY) e (brickX,brickY) rispettivamente le coordinate di uno dei punti di bordo della pallina e la posizione di un mattoncino, la seguente funzione ritorna "true" qualora il punto sia all'interno del rettangolo individuato dal brick:

```
function is_colliding ( pointX : integer; pointY : integer; brickX :
integer; brickY : integer ) return boolean is
begin
      if (pointX>=brickX and pointX<brickX+BRICK_WIDTH and
      pointY>=brickY and pointY<brickY+BRICK_HEIGHT) then
            return true;
      else
            return false;
      end if;
end is_colliding;
```

Come si può notare dall'immagine sottostante, eseguendo la funzione per ognuno degli 8 punti di bordo, è possibile discriminare quali siano all'interno del mattoncino, deducendo così la direzione di provenienza della sfera e rendendo possibile la predizione del successivo rimbalzo.



Facendo riferimento alla seguente nomenclatura:



al fine di ottenere un rimbalzo realistico, vengono prima valutati i punti *left*, *top*, *right*, e *bot*: in caso di collisione, viene invertito il segno dell'opportuna componente della velocità e viene effettuato il calcolo della simmetria rispetto ad uno dei lati del brick, in maniera assolutamente analoga a quanto viene fatto per la collisione con i bordi dell'area di gioco.

In caso di mancata collisione con i suddetti punti, vengono valutati *topLeft*, *topRight*, *botLeft* e *botRight*: se `is_colliding` ritorna "true" per uno di questi punti, significa

che la sfera ha colpito il brick in uno spigolo e il cambio di segno deve avvenire sia per la componente *x* che per quella *y* della velocità.

Si noti come alcune configurazioni non siano possibili (pallina contrassegnata con "NO" nelle prima immagine): la velocità massima della sfera, definita con la costante `BALL_MAX_SPEED` in *arkanoid_package.vhd*, è tale da garantire la mutua esclusione delle collisioni relative ai punti *left*, *top*, *right*, e *bot*, assicurando un comportamento coerente in ogni situazione.

4.6.2 IMPLEMENTAZIONE OTTIMIZZATA

Come è possibile valutare ad ogni game-logic update eventuali collisioni della sfera?
Da un rapido calcolo si evince che per individuare il rimbalzo della pallina con la matrice dei mattoncini, sono richieste in totale 150 x 8 (numero totale di brick moltiplicato per tutti i punti di bordo della sfera) = 1200 valutazioni. Si tratta di un numero troppo grande di test da effettuare in un solo ciclo di clock, a meno di non generare una rete combinatoria molto estesa.
Per questo motivo, dopo una prima realizzazione basata su singolo clock al fine di testare il corretto comportamento dell'algoritmo di collisione, si è immediatamente cercato di realizzare un'implementazione più elegante e meno onerosa in termini di porte logiche.
Ci si è subito resi conto che, poiché l'aggiornamento effettivo degli elementi della scena avviene ad ogni game-logic update (che, si ricordi, si attiva ogni 416667 clock), non vi era assolutamente la necessità di utilizzare un costoso "for" all'interno di un singolo clock, ma la valutazione delle collisioni poteva essere dilatata nei cicli di clock che intercorrono tra una fase di game-logic update e la successiva.
Dunque, non appena termina il periodo di clock corrispondente ad uno di tali update, si azzera un apposito segnale che indica l'inizio della fase di controllo delle collisioni con i brick: essa dura 150 clock, in ognuno dei quali una semplice rete valuta l'eventuale collisione della pallina con il mattoncino attualmente in esame.
Qualora essa dovesse sussistere, la fase di controllo termina ed il risultato della collisione (mattoncino distrutto o semplice rimbalzo nel caso dei mattoncini grigi o oro) viene comunicato tramite una variabile condivisa a *LevelProcess*, l'unico processo che può pilotare ed eventualmente modificare i segnali relativi alla matrice dei mattoncini.

4.7 MOVIMENTO DELLA PALLINA

4.7.1 LOGICA DI BASE

Come anticipato nell'introduzione, durante la scelta della tecnica da utilizzare per muovere la pallina, si volevano soddisfare due ipotesi:

1. Possibilità di regolare con un'ottima granularità l'angolo del vettore velocità della sfera
2. Velocità che aumenta nel tempo in maniera fluida

Le basi per soddisfare questi requisiti erano già state poste grazie all'utilizzo dello "spazio di gioco", di dimensione molto maggiore rispetto allo "spazio VGA". Non restava che implementare un algoritmo in grado di fruire nel migliore dei modi di questa potenzialità.

Come miglior implementazione è stata valutata quella che rispettava in maniera più rigorosa le leggi della fisica e si è deciso dunque di basarsi sul teorema di Pitagora.

La velocità della sfera può essere espressa come un vettore s*peedVector* in uno spazio bidimensionale, scomponibile nelle sue componenti *speedX* e *speedY*. Essendo queste ultime due vettori perpendicolari tra loro, per il teorema di Pitagora si potrà scrivere:

$speedX^2 + speedY^2 = speedVector^2$

Andiamo ora ad osservare come questo concetto venga utilizzato per garantire le ipotesi iniziali.

4.7.2 MOTIVAZIONE TEORICA

Riprendiamo l'equazione appena vista e ricordiamo come ogni unità di misura sia da considerarsi moltiplicata per UNIT (= 16) nell'ambito dello spazio di gioco.
Sia $totalSpeed^2$ il quadrato del modulo del vettore velocità attuale. Consideriamo ora $totalSpeed^2$ "separato" dal fattore UNIT, ottenendo $partialSpeed^2 \times UNIT^2$. Sia ora λ un numero intero tale che $0 \leq \lambda \leq UNIT$. È possibile scrivere l'uguaglianza:

$$\lambda \times partialSpeed^2 \times UNIT + (UNIT - \lambda) \times partialSpeed^2 \times UNIT = partialSpeed^2 \times UNIT^2$$

Ciò significa che, scelta una velocità *target* (*totalSpeed*^{2}), è possibile calibrare un parametro λ, con un valore da 0 a 16, al fine di ripartire il modulo velocità totale in due componenti differenti, dove:

- $\lambda \times partialSpeed^2 \times UNIT$ rappresenta $ballSpeedX^2$
- $(UNIT - \lambda) \times partialSpeed^2 \times UNIT$ rappresenta $ballSpeedY^2$

Effettuando la radice quadrata delle due velocità, si ottengono le componenti orizzontali e verticali, che rispettano il teorema di Pitagora (ipotenusa = modulo vettore velocità totale).

4.7.3 IMPLEMENTAZIONE

Facendo riferimento all'ultima equazione mostrata, $partialSpeed^2 \times UNIT$ viene gestito da *SpeedProcess*, che ogni 15000000 incrementa di 1 il suo valore, salvandolo nel segnale *squaredSpeed*.
λ e (*UNIT* – λ) sono rappresentati rispettivamente da *ballAngleX* e *ballAngleY* (la cui somma è sempre 16), che di fatto indicano la direzione del vettore velocità e vengono modificati in concomitanza con il rimbalzo contro il paddle (unico tipo di collisione che non fa variare soltanto il segno).

Al termine di ogni game-logic update, vengono presentati all'ingresso dei moduli per il calcolo della radice quadrata i seguenti valori:

```
SQUARED_X<=std_logic_vector(to_unsigned(squaredSpeed*ballAngleX,
SQUARED_X'length));

SQUARED_Y<=std_logic_vector(to_unsigned(squaredSpeed*ballAngleY,
SQUARED_Y'length));
```

congiuntamente all'attivazione di un apposito segnale di CLEAR, attivo per un ciclo di clock.
Nel momento in cui CLEAR si disabilita, inizia il calcolo della radice quadrata, che impiega 16 cicli di clock, sfruttando lo *square root abacus algorithm* (C. Woo).
Una prima implementazione di test prevedeva il calcolo della radice quadrata tramite una complessa rete combinatoria in grado di trovare il risultato in un singolo ciclo di clock (*Non-Restoring Square Root algorithm*, Yamin Li & Wanming Chu): tale

soluzione è stata però presto sostituita da quella definitiva, dal momento che i nuovi valori di velocità devono essere pronti entro la fase di game-logic successiva (oltre 400000 clock dopo) e non vi è dunque la necessità di effettuare il calcolo in un singolo clock. La nostra implementazione dell'*abacus algorithm* ha permesso di ridurre drasticamente la complessità dell'hardware inferito per eseguire il calcolo.

All'inizio della successiva game-logic, potranno quindi essere effettuati i seguenti assegnamenti relativi alla posizione predetta della pallina:

```
X<=ballX+to_integer(unsigned(ROOT_X))*signX;
Y<=ballY+to_integer(unsigned(ROOT_Y))*signY;
```

certi del fatto che ROOT_X e ROOT_Y (segnali di uscita dei moduli per la radice quadrata) contengono già da molto tempo i valori corretti.

4.8 COLLISIONE CON IL PADDLE

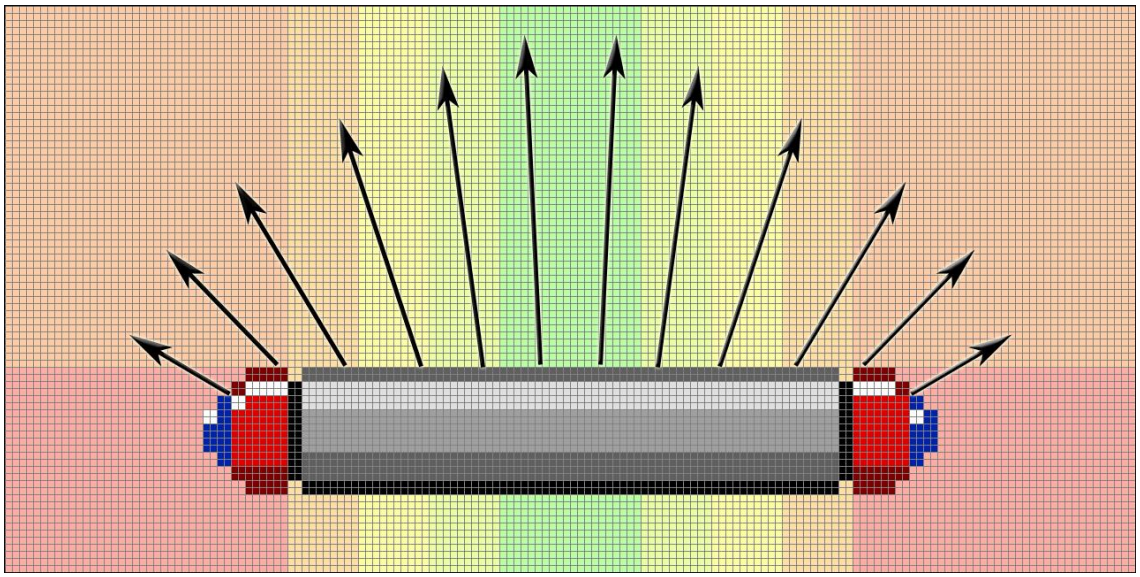
L'algoritmo per la collisione con il paddle sfrutta il sistema di velocità appena esposto, modificando i valori di *ballAngleX* e *ballAngleY*. L'angolo di uscita della sfera è funzione della posizione della pallina rispetto alla posizione del paddle nel momento in cui essa colpisce la sua superficie.

Inizialmente si era tentato un approccio lineare: più il punto di collisione era distante dal punto medio della lunghezza del paddle, più l'angolo di uscita era significativo.
Tale approccio, seppur elegante e poco oneroso, non si è rivelato però una buona soluzione in termini di giocabilità.

Si è dunque deciso di adottare una soluzione maggiormente "controllata", impostando angolazioni arbitrarie sulla base di "settori di collisione".

La parte centrale (grigia) è stata divisa in 8 zone differenti, ognuna caratterizzata da valori di *ballAngleX* e *ballAngleY*, che vengono applicati qualora la sfera colpisca il pad in tale settore.
Anche le parti laterali (rosse) sono state suddivise in due sezioni, alle quali corrispondono angolazioni risultanti più o meno accentuate.

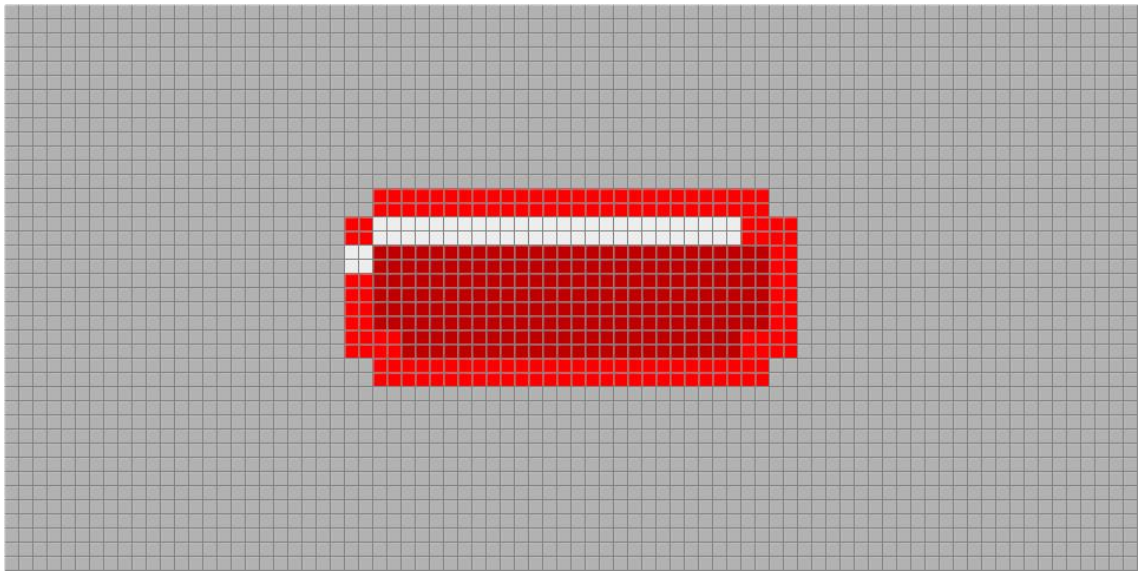
Nella seguente immagine sono evidenziate le zone di collisione per colore, da quella che causa l'angolo minore (verde) all'angolo maggiore (rosso):



4.9 POWERUP

4.9.1 RAPPRESENTAZIONE

I Powerup sono rappresentati da rettangoli 32x14 e ve ne può essere al massimo uno presente sulla scena. Essi possono comparire con una certa probabilità in seguito alla distruzione di un mattoncino e garantiscono bonus/malus al giocatore qualora venissero "catturati" dal paddle.



I cinque tipi possibili sono:

- **Life** (blu): aumenta di uno le vite del giocatore, fino ad un massimo di 4
- **Enlarge** (giallo): aumenta la larghezza del pad, facilitando la collisione con la pallina
- **Slow** (verde): rallenta la sfera
- **Fast** (rosso): aumenta la velocità della sfera
- **Death** (grigio): diminuisce di uno le vite del giocatore

4.9.2 GENERAZIONE CASUALE

Il Powerup vengono gestiti all'interno di *PowerupProcess*: ad ogni update della game-logic, il processo controlla se un mattoncino è stato distrutto nell'update precedente e, in caso affermativo, valuta se generare o meno un powerup nella posizione del brick eliminato. Questa valutazione avviene sulla base di un contatore a 5 bit che ad ogni clock aumenta di uno il suo valore interno: se la configurazione del contatore è uguale ad esempio a quella relativa al powerup "enlarge", corrispondente a '"00010", tale potenziamento viene generato ed inizia a scendere verso il fondo dello schermo con velocità costante POWERUP_SPEED_Y. Probabilità differenti sono ottenute, come nel caso del powerup "Fast", assegnando più configurazioni, tra le 32 possibili, ad un determinato tipo di potenziamento. Nel complesso, 7 configurazioni su 32 generano un powerup, mentre le restanti 25 non danno origine a nessun bonus. Questo sistema molto elementare di generazione casuale è stato giudicato sufficiente per lo scopo per il quale è stato implementato e test empirici su numerosi mattoncini distrutti hanno dimostrato che le proporzioni di probabilità sono rispettate su grandi numeri.

4.9.3 APPLICAZIONE DEGLI EFFETTI

Se è vero che è *PowerupProcess* a pilotare i segnali relativi ai powerup, è anche vero che gli effetti di questi ultimi portano a modificare segnali che non sono pilotati da tale processo. Analizziamo quali processi pilotano i *signal* che devono essere modificati in concomitanza con la cattura dei diversi powerup:

- **Life**: *BallProcess*
- **Enlarge**: *PaddleProcess*

- **Slow**: *SpeedProcess*
- **Fast**: *SpeedProcess*
- **Death**: *BallProcess*

Per ovviare alla mancata possibilità da parte di *PowerupProcess* di modificare a piacimento i segnali relativi a pallina, paddle e velocità della sfera, è stato adottato il seguente sistema: un segnale all'interno del *Datapath* di tipo enumerativo chiamato powerUpCaught (valori possibili: P_NULL, P_LIFE, P_SLOW, P_FAST, P_ENLARGE, P_DEATH), con valore di default P_NULL, viene posto al valore corrispettivo al powerup appena catturato per un singolo periodo di clock. Nel clock successivo a questa commutazione, i processi *BallProcess*, *PaddleProcess* e *SpeedProcess* controllano se powerUpCaught ha assunto un valore che rientra nel loro dominio di azione e, in caso affermativo, modificano opportunamente i valori dei segnali dei quali solo essi hanno il controllo.

4.10 SUONI

Al fine di riprodurre suoni in concomitanza con determinati eventi del gioco (rimbalzo della pallina contro il pad, distruzione di un mattoncino, …) è stato realizzato un apposito modulo *arkanoid_sound*. Esso sfrutta il pin GPIO_0(0) della board per inviare onde quadre a diverse frequenze ad un *buzzer* esterno; non si è ritenuto necessario utilizzare il chip *Wolfson WM8731* presente all'interno della scheda, considerata la semplicità dei toni richiesti.

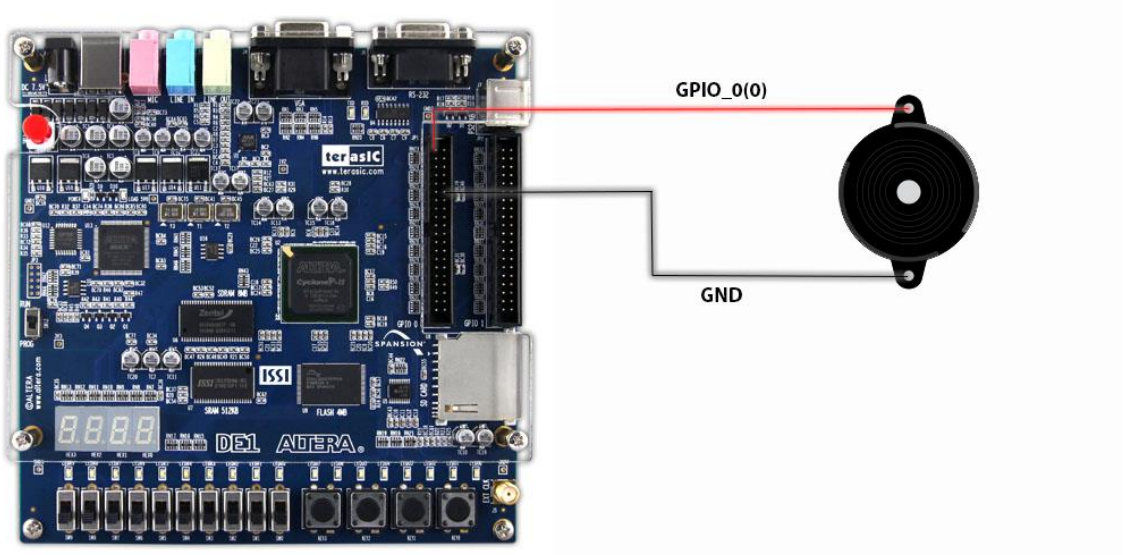


Figura – Connessioni tra la board e il buzzer

Sono previsti 3 toni diversi:

- **SoundBound** *(Si3)*: eseguito quando la pallina colpisce un bordo dell'area di gioco
- **SoundPaddle** *(Si4)*: un'ottava più alto di *SoundBound*, emesso in concomitanza con la collisione con il paddle
- **SoundBrick** *(Fa#5)*: intervallo di quinta rispetto a *SoundPaddle*, udibile quando la sfera colpisce un mattoncino

Quando si scatena un evento che causa l'emissione di un suono, *BallProcess* attiva un apposito segnale (SOUND_CODE), attivo per un solo clock, indirizzato al componente *arkanoid_sound*. Alla ricezione, quest'ultimo imposta il contatore counterSound relativo al suono da eseguire al suo valore massimo: questo contatore inizierà a decrementare il suo valore ad ogni clock fino a raggiungere 0, in un tempo pari a circa 1/12 di secondo; durante tale finestra temporale, viene inviata al pin GPIO_0(0) (rinominato SOUND_PIN) l'onda quadra necessaria affinché il *buzzer* emetta il tono desiderato.
Le onde quadre relative ai vari suoni vengono costantemente generate (ma non necessariamente propagate a SOUND_PIN) sfruttando contatori discendenti che "ripartono" sempre da valori che dipendono dalla frequenza della nota desiderata e, ad ogni raggiungimento dello 0, negano il valore corrente del segnale SOUND_PIN.
Ad esempio, per "sound paddle", il valore di ripartenza del contatore per la generazione dell'onda quadra è dato da:

```
CLOCK_FREQUENCY / SOUND_PADDLE_FREQUENCY /2 - 1
```

Dove CLOCK_FREQUENCY è pari a 50000000 (frequenza di clock, cioè cicli di clock in un secondo) e SOUND_PADDLE_FREQUENCY è la frequenza in *Hz* della nota *Si4* (494). Dividendo per due tale frazione, si ottiene la metà del periodo dell'onda necessaria per produrre il suono desiderato.

Capitolo 5. VIEW

5.1 STRUTTURA

La *View* ha lo scopo di mostrare a video gli elementi del gioco, sulla base delle informazioni ricevute dai moduli *Datapath* e *Controller*.
La sua struttura interna è divisa in 3 *process*:

- **DrawProcess**: process principale, adibito alla gestione del protocollo VGA e alla rappresentazione degli oggetti nella scena
- **AnimationProcess**: macchina a stati che controlla le animazioni degli elementi dinamici del livello
- **Animation_time_generator**: genera il segnale di "cambio stato animazione"

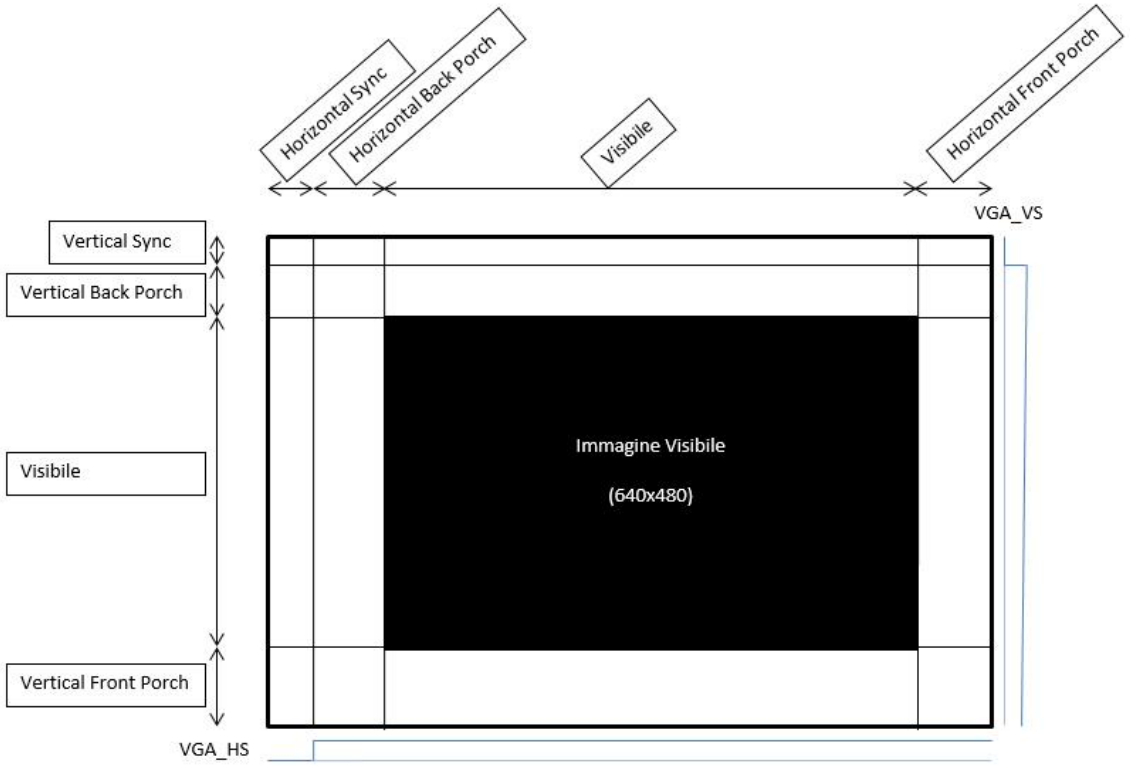
Il modulo *View* utilizza le costanti e le funzioni definite in *vga_package*, relative principalmente ai timing richiesti dalla VGA e ai colori dei diversi oggetti.

5.2 PROTOCOLLO VGA

Si è scelto di utilizzare lo standard 640x480 @60Hz per quanto concerne la VGA. Questo tipo di configurazione richiede determinati *timing* nell'ambito del protocollo dedicato:

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31

Il Pixel Clock giunge al modulo dal PLL descritto nella prima parte di questa trattazione e viene instradato verso l'apposito PIN della porta VGA presente sulla scheda.
I numeri esposti in tabella sono da riferire all'immagine sottostante, che chiarisce il significato di ogni valore:



	Horizontal Sync	Horizontal Back Porch	Image	Horizontal Front Porch
VGA_HS	0	1	1	1

	Vertical Sync	Vertical Back Porch	Image	Vertical Front Porch
VGA_VS	0	1	1	1

Il componente controlla la porta VGA attraverso 3 bus di 4 bit per i canali colore e 2 segnali per la sincronizzazione (verticale e orizzontale).

Per visualizzare un frame di 640x480 pixel a 60Hz utilizzando il protocollo VGA è necessario che il componente lavori ad una frequenza di 25MHz e che rispetti i tempi di sincronizzazione di back e front porch (sia verticali che orizzontali, esposti in tabella). Quindi, a fini pratici, ogni schermata da visualizzare ha:

- larghezza pari a `H_FRONT_PORCH + H_SYNC_PULSE + H_BACK_PORCH + VISIBLE_WIDTH` = 800 pixel
- altezza pari a `V_FRONT_PORCH + V_SYNC_PULSE + V_BACK_PORCH + VISIBLE_HEIGHT` = 525 pixel

Due segnali *x* ed *y* percorreranno dunque tutta la griglia 800x525 e i bus relativi ai colori RGB saranno impostati ai valori desiderati all'interno dell'area visibile, mentre dovranno essere posti a 0 (nero) nelle aree "nascoste".
Per ottenere questo comportamento è necessario individuare l'area visibile: ciò viene

fatto tramite lo *statement condizionale*

```
x>=WINDOW_HORIZONTAL_START and x<WINDOW_HORIZONTAL_END and
y>=WINDOW_VERTICAL_START and y<WINDOW_VERTICAL_END
```

dove le costanti che compaiono sono date da

- **WINDOW_HORIZONTAL_START** := H_FRONT_PORCH + H_SYNC_PULSE; --112
- **WINDOW_HORIZONTAL_END** := H_FRONT_PORCH + H_SYNC_PULSE + VISIBLE_WIDTH; --752
- **WINDOW_VERTICAL_START** := V_FRONT_PORCH + V_SYNC_PULSE; --12
- **WINDOW_VERTICAL_END** := V_FRONT_PORCH + V_SYNC_PULSE + VISIBLE_HEIGHT;--492

ed individuano una finestra di 640x480 pixel, entro la quale verranno disegnati gli oggetti.

5.3 RAPPRESENTAZIONE DEGLI OGGETTI

Sulla base della posizione individuata da *x* ed *y*, è possibile assegnare ai 16 bit relativi ai colori RGB (4 per ogni canale, 4096 colori possibili) diversi valori, relativamente all'oggetto che si desidera disegnare a video.

Le coordinate degli elementi della scena arrivano alla VGA già in formato "spazio VGA", poiché la divisione per UNIT avviene in *arkanoid.vhd*, precisamente nel *port map* della entity *View*.

Viene ora riportato il codice per la rappresentazione della pallina (spiegazioni nei commenti):

```
--Qualora x ed y siano all'interno del rettangolo 13x13 che contiene
--la sfera…
if(x>=BALL_X and x<BALL_X+BALL_SIZE/UNIT and y>=BALL_Y and
y<BALL_Y+BALL_SIZE/UNIT) then
      --disegno del bordo di colore scuro
      if
      (
      (x>=BALL_X+4 and x<BALL_X+9 and y>=BALL_Y+0 and y<BALL_Y+13) or
      (x>=BALL_X+2 and x<BALL_X+11 and y>=BALL_Y+1 and y<BALL_Y+12) or
      (x>=BALL_X+1 and x<BALL_X+12 and y>=BALL_Y+2 and y<BALL_Y+11) or
      (x>=BALL_X+0 and x<BALL_X+13 and y>=BALL_Y+4 and y<BALL_Y+9)
      )
      then
            VGA_R <= COLOR_BALLBORDER(0 to 3);
            VGA_G <= COLOR_BALLBORDER(4 to 7);
            VGA_B <= COLOR_BALLBORDER(8 to 11);
      end if;
      --riempimento dell'interno con un grigio molto chiaro.
```

```
        --Per il comportamento dei signal, se le condizioni di questo
        --"if" sono verificate, i seguenti assegnamenti dei bus RGB
        --della VGA sostituiscono gli assegnamenti precedenti
        if
        (
        (x>=BALL_X+5 and x<BALL_X+8 and y>=BALL_Y+2 and y<BALL_Y+11) or
        (x>=BALL_X+3 and x<BALL_X+10 and y>=BALL_Y+3 and y<BALL_Y+10) or
        (x>=BALL_X+2 and x<BALL_X+11 and y>=BALL_Y+5 and y<BALL_Y+8)
        )
        then
              VGA_R <= COLOR_BALLFILL(0 to 3);
              VGA_G <= COLOR_BALLFILL(4 to 7);
              VGA_B <= COLOR_BALLFILL(8 to 11);
        end if;
end if;
```

Tutti gli altri elementi vengono disegnati seguendo la stessa logica.

Lo sfondo a quadretti di colore alternato è stato realizzato sfruttando due *flag* (backgroundFlagX e backgroundFlagY), ai quali viene assegnato il proprio valore negato ogni N pixel (con N = lato dei quadrati), utilizzando l'operatore modulo. È dunque sufficiente valutare la veridicità della condizione (backgroundFlagX = backgroundFlagY) ed assegnare conseguentemente ai pin RGB i valori corrispondenti al blu chiaro o al blu scuro per ottenere uno sfondo stile "scacchiera".

5.4 ANIMAZIONI

Le animazioni, gestite da *AnimationProcess* e realizzate grazie ad una macchina a stati di Moore, interessano le luci laterali del paddle e i cuori che simboleggiano le vite rimanenti.

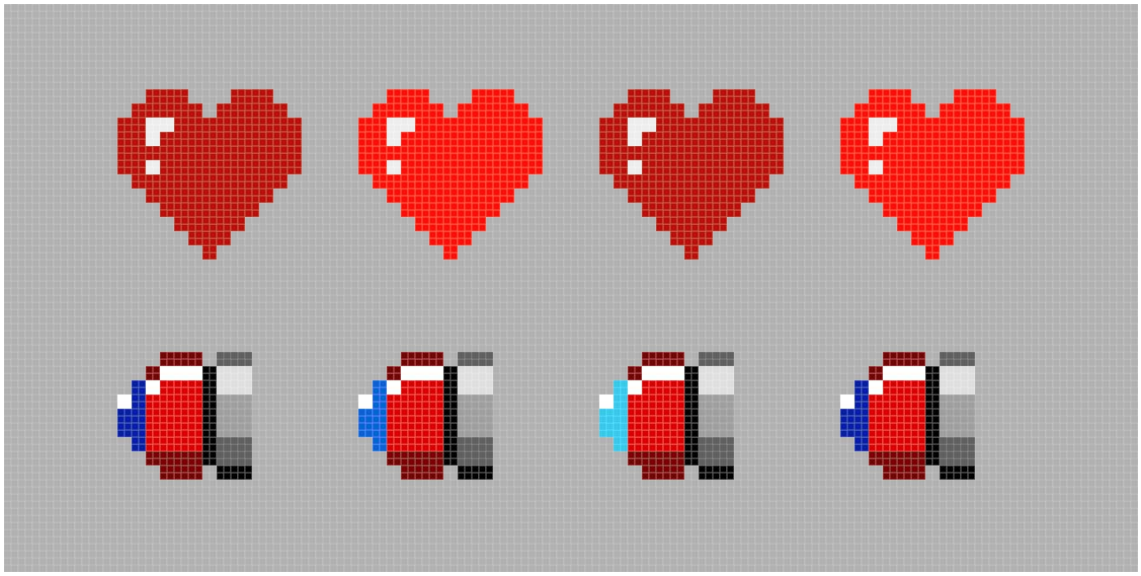
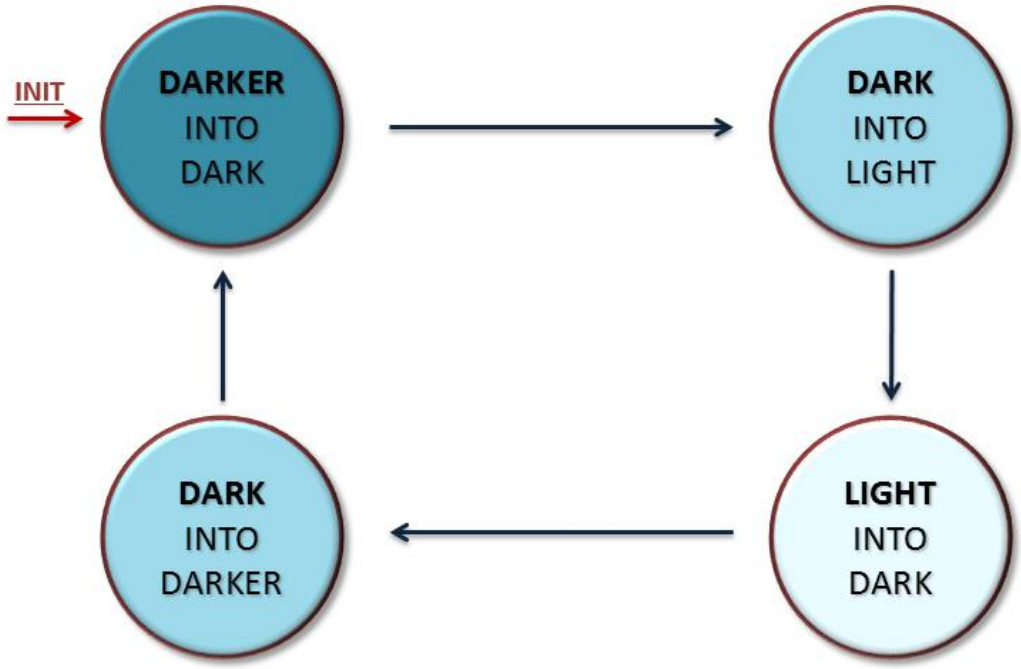


Figura – Le quattro fasi delle animazioni

In figura si possono notare i quattro stati dell'animazione (in alto i cuori, in basso le luci del pad).

Lo stato commuta ogniqualvolta il segnale changeLight assume il valore '1', evento controllato da *animation_time_generator* e che si verifica ogni 10000000 clock. La macchina a stati che ne deriva è la seguente:



La stessa macchina a stati è utilizzata per controllare sia il colore delle luci del pad che quello dei cuori, sebbene per questi ultimi gli stati effettivi siano solo due.

5.5 SCHERMATE DI GIOCO (IMMAGINI)

Sono infine riportate alcune immagini tratte dal prodotto finito.

Per una dimostrazione più dettagliata nella quale è possibile osservare la fase di gameplay e le tutte le *feature* introdotte (suoni, animazioni, keyboard, …), si rimanda al **video**, appositamente realizzato, reperibile presso il seguente link (ve ne è una copia a bassa definizione anche nella directory contenente questa relazione):

https://youtu.be/xByfgy9wtiA