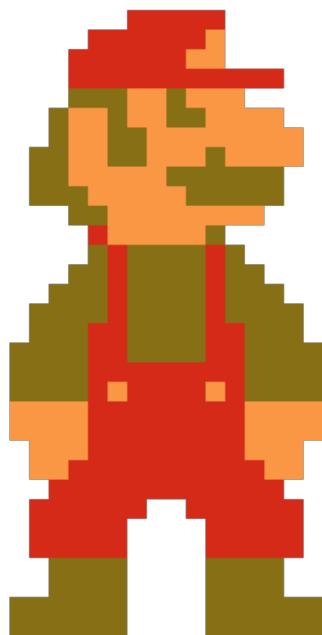


ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA
PROGETTO FINALE ED ATTIVITÀ PROGETTUALE
DI
Sistemi Digitali M
Mario VHDL
per
Altera DE1 Board



Marco Bertolazzi
0000884790

Mattia Sinigaglia
0000883967

Fabrizio Torriano

-

Indice

INTRODUZIONE	3
SUPER MARIO BROS.....	4
LA NOSTRA VERSIONE	5
ARCHITETTURA	6
TOP-LEVEL ENTITY.....	7
PACKAGE	8
PLL	9
SNES CONTROLLER.....	10
<i>Protocollo SNES</i>	11
<i>mario_snes</i>	13
CONTROL UNIT	15
<i>ActionProcess e GameClockProcess</i>	16
<i>StateProcess</i>	17
<i>GameTimingProcess</i>	20
DATAPATH.....	21
<i>mario_datapath</i>	24
<i>movement_machine</i>	26
<i>collision_machine</i>	29
<i>memory_mapper</i>	31
<i>sram_controller</i>	33
VIEW.....	34
<i>mario_view</i>	36
AUDIO.....	41
CODEC AUDIO WM8731.....	42
<i>i2c</i>	42
i2cClockProcess	43
FSMI2CProcess	44
InitProcess	46
<i>audio_mario</i>	47
AudioGenProcess.....	48
ReadAudioProcess	49
BUZZER.....	50
FONTI	51

Introduzione

Nel 1981, dalla fantasia del disegnatore di videogame Shigeru Miyamoto, nasce un piccolo eroe coi baffi che ha la missione di salvare la sua fidanzata Pauline imprigionata dal terribile scimmione Donkey.

Pauline invoca l'aiuto dell'impavido uomo in tuta da lavoro: Jumpman.

Quattro anni dopo viene rilanciato da [Nintendo](#) sotto il nome di Super Mario. In questi 34 anni, il piccolo idraulico di origini italiane, di strada ne ha fatta tanta, diventando uno dei videogiochi più venduti di tutti i tempi e apparendo in oltre 100 videogame.

1985 Super Mario Bros.	1989 Super Mario Bros. 2	1992 Super Mario	1997 Super Mario 64	2002 Super Mario Sunshine	2007 Super Mario Galaxy	2009 New Super Mario Bros. Wii	2014 Mario Kart 8	2015 Super Mario Maker	2015 Super Mario Odyssey

Super Mario Bros

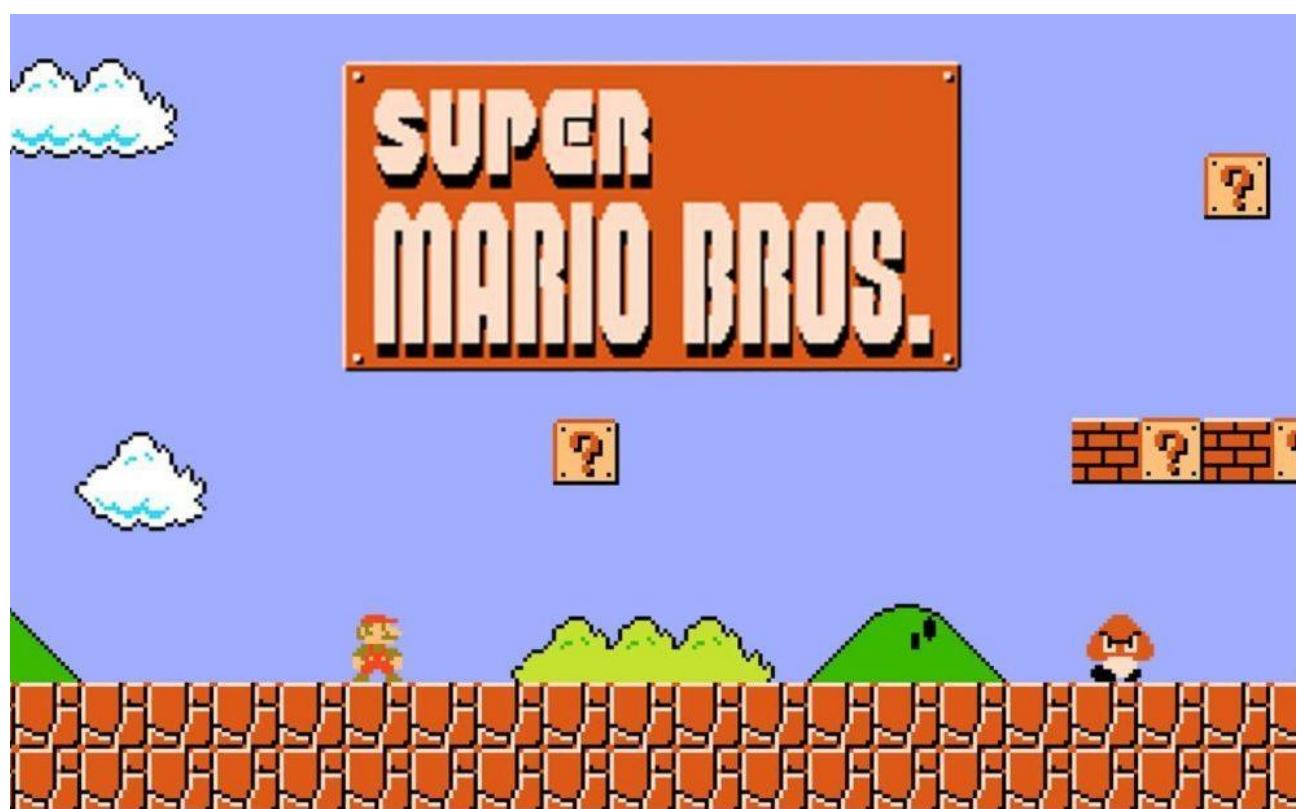
In Super Mario Bros il giocatore controlla i movimenti e le azioni di Mario tramite il controller SNES (Super Nintendo Entertainment System).

L'obiettivo del gioco è attraversare i tanti mondi del Regno dei Funghi, ciascuno suddiviso in livelli, e salvare la Principessa.

In Super Mario Bros le monete sono gli oggetti più comuni del gioco, esse si trovano in qualunque livello e se si raggiungono le 100 monete catturate, il personaggio guadagna una vita.

Se invece Mario esaurisce il tempo massimo per il completamento del livello perde una vita e ricomincia in tal caso il livello dall'ultimo checkpoint raggiunto.

Per completare il gioco, il giocatore deve superare otto mondi, ognuno suddiviso in quattro livelli per un totale di 32 livelli.



La nostra versione

Ispirati dalla versione originale di Super Mario Bros e da altri videogiochi, abbiamo voluto creare una versione personale in cui Mario si potesse muovere nell'area di gioco per acchiappare le monete.

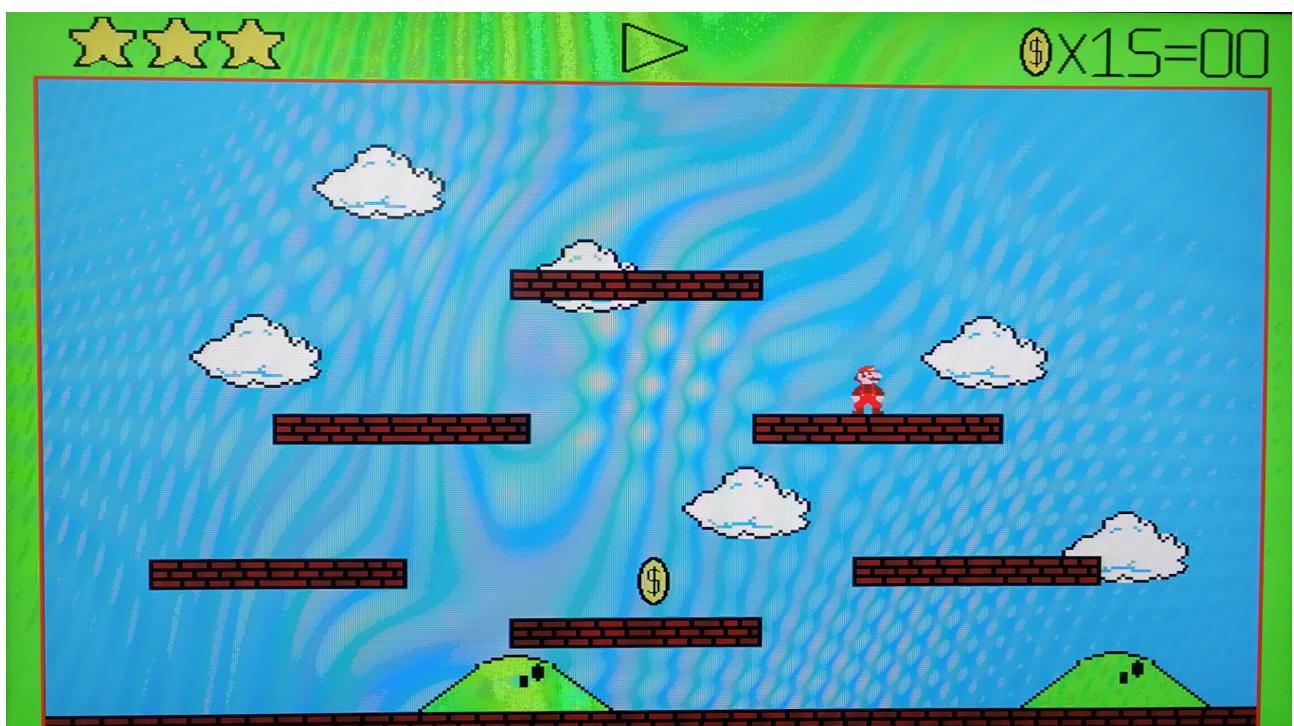
In particolare, il giocatore deve guidare Mario facendolo saltare sui blocchi sospesi per aria e fargli catturare tutte le monete entro un tempo limite.

Il gioco è composto da 5 livelli di difficoltà crescente: i blocchi sono posti in posizioni sempre più difficili da raggiungere e il tempo a disposizione per catturare una moneta diminuisce gradualmente.

Per rendere più competitiva la partita, Mario ha un numero di vite limitato a 3 e ogni volta che vince un livello ne ottiene una in più. In caso contrario il numero di vite decresce.

Infine, se Mario perde tutte le vite, il gioco ricomincia dal primo livello.

Per un'esperienza di gioco più coinvolgente, il giocatore controlla Mario attraverso il controller SNES originale.



È disponibile su YouTube una breve presentazione del nostro Super Mario FPGA al seguente link: <https://www.youtube.com/watch?v=ItshXTzq0hM>

Architettura

L'intera architettura è organizzata secondo il pattern MVC:

- **Datapath**

Gestisce un insieme di dati logicamente correlati, riceve dalla *Control Unit* l'input dell'utente e comunica alla *View* le posizioni degli elementi all'interno della scena. Inoltre, interagisce con la *Control Unit* al fine di inviare informazioni relative all'avanzamento del gioco e di rispondere correttamente agli eventuali cambiamenti di stato del sistema.

- **View**

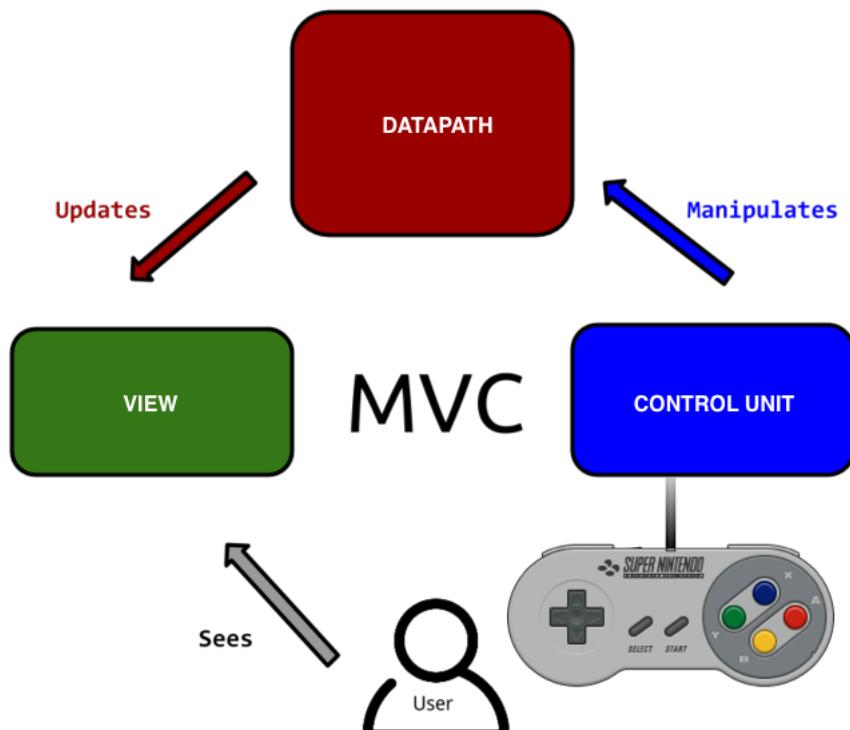
Gestisce un'area di visualizzazione sulla quale presenta all'utente i dati ricevuti dal *Datapath*.

L'output di visualizzazione utilizzato è lo standard VGA con risoluzione 640x480.

- **Control Unit**

Gestisce l'avanzamento di stato del sistema e tramite opportuni segnali di controllo regola le attività svolte da *Datapath* e *View*.

È inoltre responsabile della gestione dell'input dell'utente proveniente dal modulo *mario_snes* che implementa il protocollo del controller SNES.



Top-Level Entity

La top-level entity rappresenta il livello più alto della gerarchia e si interfaccia direttamente con la board Altera DE1.

Dato che questa è stata la nostra prima esperienza nell'utilizzo di una FPGA così complessa, abbiamo voluto cogliere l'occasione per poter imparare a gestire ed usare le diverse features messe a disposizione della board.

Di seguito la top level entity e una breve descrizione delle features utilizzate:

```
entity mario is
  port(
    CLOCK_50      : in std_logic;
    KEY           : in std_logic_vector(3 downto 0);

    --GPIO FOR SNES CONTROLLER AND ACTIVE BUZZER
    GPIO_0         : out std_logic_vector(35 downto 0);
    GPIO_1         : in std_logic_vector(35 downto 0);

    LEDR          : out std_logic_vector(9 downto 0);
    LEDG          : out std_logic_vector(6 downto 0);
    SW             : in std_logic_vector(9 downto 0);

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

    --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;

    --CODEC WM8731
    AUD_BCLK       : out std_logic;
    AUD_XCK        : out std_logic;
    AUD_DACLRCK    : out std_logic;
    AUD_DACDAT    : out std_logic;
    I2C_SCLK       : out std_logic;
    I2C_SDAT       : inout std_logic;

    --FLASH
    FL_ADDR        : out std_logic_vector(21 downto 0);
    FL_DQ          : in std_logic_vector(7 downto 0);
    FL_OE_N        : out std_logic;
    FL_RST_N       : out std_logic;
    FL_WE_N        : out std_logic
  );
end;
```

- **GPIO**

Interfacciamento con il controller SNES e Buzzer attivo;

- **SRAM**

Mappatura del livello corrente;

- **VGA**

Visualizzazione su monitor esterno con risoluzione 640x480;

- **CODEC WM8731**

Riproduzione audio su altoparlanti in uscita;

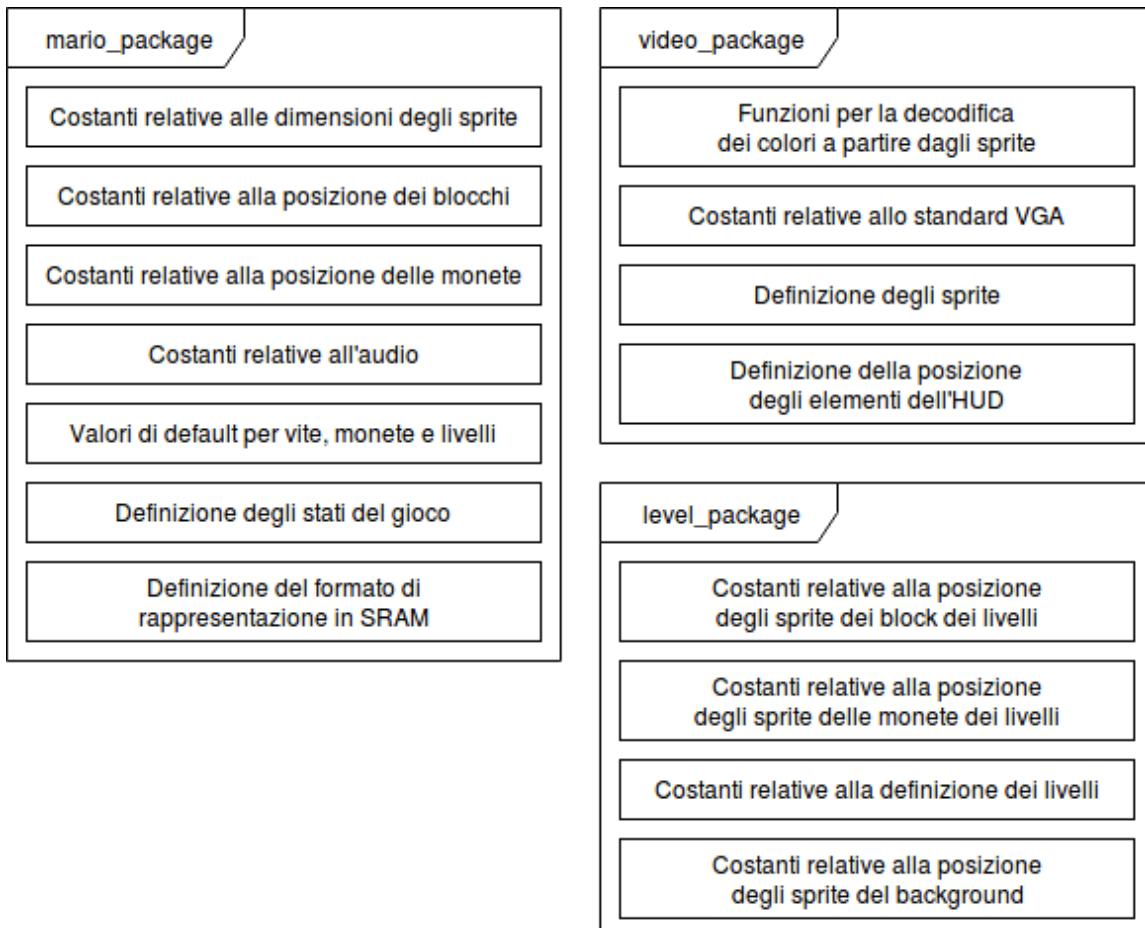
- **FLASH**

Memorizzazione traccia audio;

Package

I package definiti contengono tutte le costanti e le funzioni di utilità necessarie al funzionamento del progetto.

Tali costanti, in congiunzione con l'utilizzo dei moduli parametrizzati, permettono una maggiore scalabilità del sistema creato e una maggiore facilità di modifica.

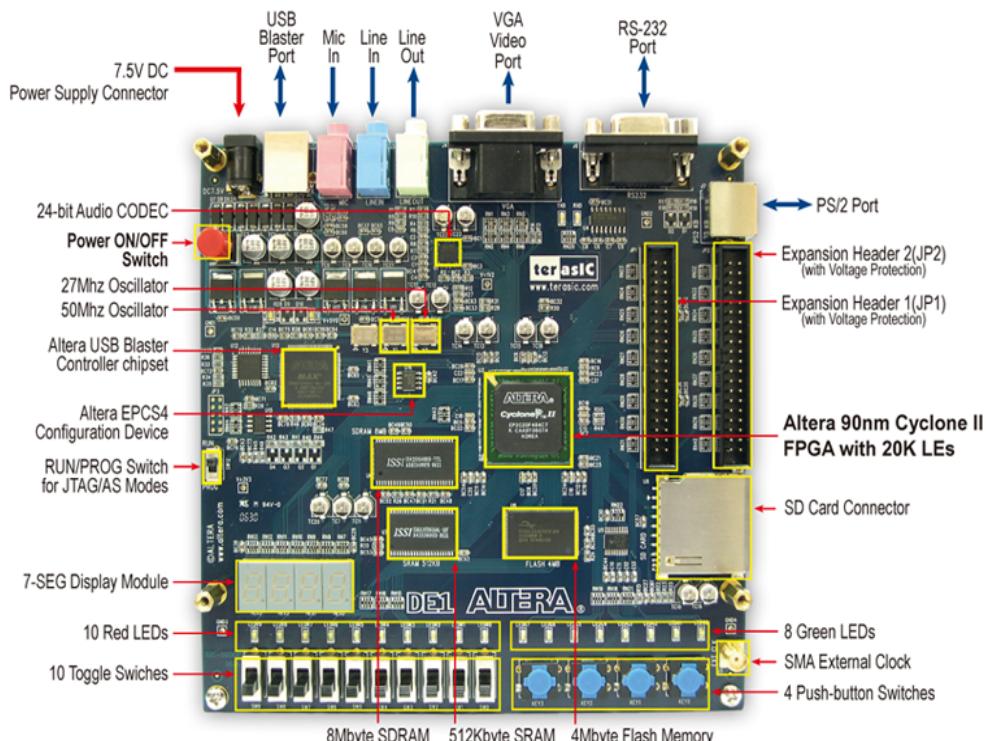


PLL

Il progetto utilizza un *PLL* (Phase-Locked Loop) per la generazione dei tre clock utilizzati dall'intero sistema.

Lo scopo principale di un PLL è quello di sincronizzare la fase e la frequenza di un segnale generato da un oscillatore di riferimento per produrre i clock dell'intero sistema. Ogni dispositivo Cyclone II ha fino a quattro PLL, supportando funzionalità avanzate come la commutazione del clock e la commutazione programmabile.

Questi PLL offrono moltiplicazione e divisione del clock, spostamento di fase e duty-cycle programmabile.



Il PLL è così definito:

```
pll : entity work.PLL
port map
(
    inclk0      => CLOCK_50,
    c0          => clock,
    c1          => clockVGA,
    c2          => clockAUDIO
);
```

Il PLL riceve in ingresso il segnale *CLOCK_50* che rappresenta il segnale di riferimento generato dall'oscillatore a 50MHz.

Il segnale *clock* ha una frequenza 50MHz come il segnale di riferimento *CLOCK_50* e rappresenta il segnale di sincronismo principale del sistema.

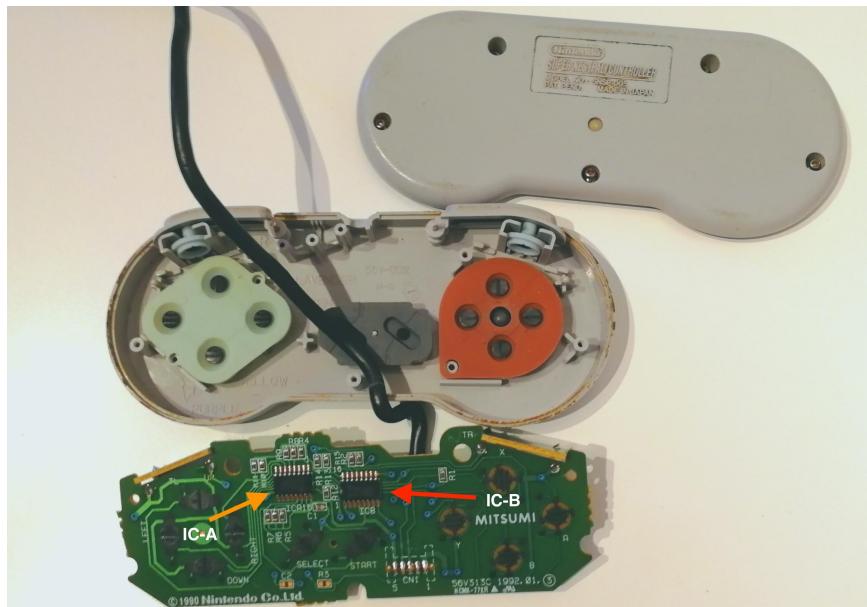
Il segnale *clockVGA* rappresenta il Pixel Clock, ha frequenza 25MHz ed è il segnale di sincronismo usato dai moduli che implementano la *View*.

Il segnale *clockAUDIO* invece ha frequenza 12MHz ed è usato dai moduli per il controllo del CODEC WM8731.

SNES Controller

Il controller sviluppato da Nintendo è costituito da un PCB avente 2 componenti a 16 pin contrassegnati come IC-A e IC-B collegati in serie.

I due componenti IC sono degli Shift Register a 8 bit, nello specifico IC-B è collegato in serie ad IC-A che porta in uscita il dato.



Il connettore del controller SNES presenta i seguenti pinout:

N° Pin	Descrizione	IN/OUT	Colore
1	+5V	IN	BIANCO
2	DATA CLOCK	IN	GIALLO
3	DATA LATCH	IN	ARANCIO
4	SERIAL DATA	OUT	ROSSO
5	-	-	-
6	-	-	-
7	GND	OUT	NERO



Protocollo SNES

Per iniziare lo scambio dati fra il controller e l'utilizzatore, bisogna inviare al controller SNES sul pin 3 (vedi tabella pinout) un impulso positivo di $12\mu s$.

In questo modo si chiede agli IC nel controller di campionare lo stato di tutti i pulsanti. Sei microsecondi dopo il fronte di discesa dell'impulso sul pin 3, il controller SNES si aspetta di ricevere sul pin 2, 16 impulsi di clock con duty cycle 50% e periodo di $12\mu s$. A questo punto il controller propaga in uscita sul pin 4 lo stato dei pulsanti campionato da IC-A e IC-B in fase di attivazione del pin 3.

Nello specifico, il controller propaga in uscita il dato al fronte di salita del clock per essere correttamente letto dall'utilizzatore nel secondo semiperiodo in cui si ha il fronte di discesa.

Ad ogni pulsante sul controller viene assegnato un ID che corrisponde all'i-esimo ciclo di clock durante il quale verrà segnalato lo stato di quel pulsante.

Il segnale propagato sul pin 4 è in logica negativa, quindi se all'i-esimo clock è registrato premuto il pulsante con ID=i allora il valore logico sul pin 4 sarà pari a 0, 1 in caso contrario.

Clock Cycle - ID	Pulsante
1	B
2	Y
3	SELECT
4	START
5	UP
6	DOWN
7	LEFT
8	RIGHT
9	A
10	X
11	L
12	R
13-16	-

Alla fine della sequenza dei 16 cicli di clock, la linea dati sul pin 4 è impostata al valore logico 0 fino al successivo impulso di Latch sul pin 3 che arriverà $6\mu s$ dopo i 16 impulsi di clock.

L'unica deviazione dal funzionamento appena descritto si verifica nel primo ciclo di clock in cui si deve propagare lo stato registrato per il pulsante B avente ID=1.

Il clock è normalmente ad un livello logico 1, quindi la prima transizione che viene effettuata dopo il segnale di latch è un fronte di discesa.

In questa particolare situazione, il controller anticipa la propagazione del segnale sul pin 4 al fronte di discesa del clock, per tutti gli altri pulsanti la propagazione avviene sul fronte positivo come descritto.

Per maggiore chiarezza è presentato un esempio di forme d'onda del protocollo realizzato con il software LTSpice.

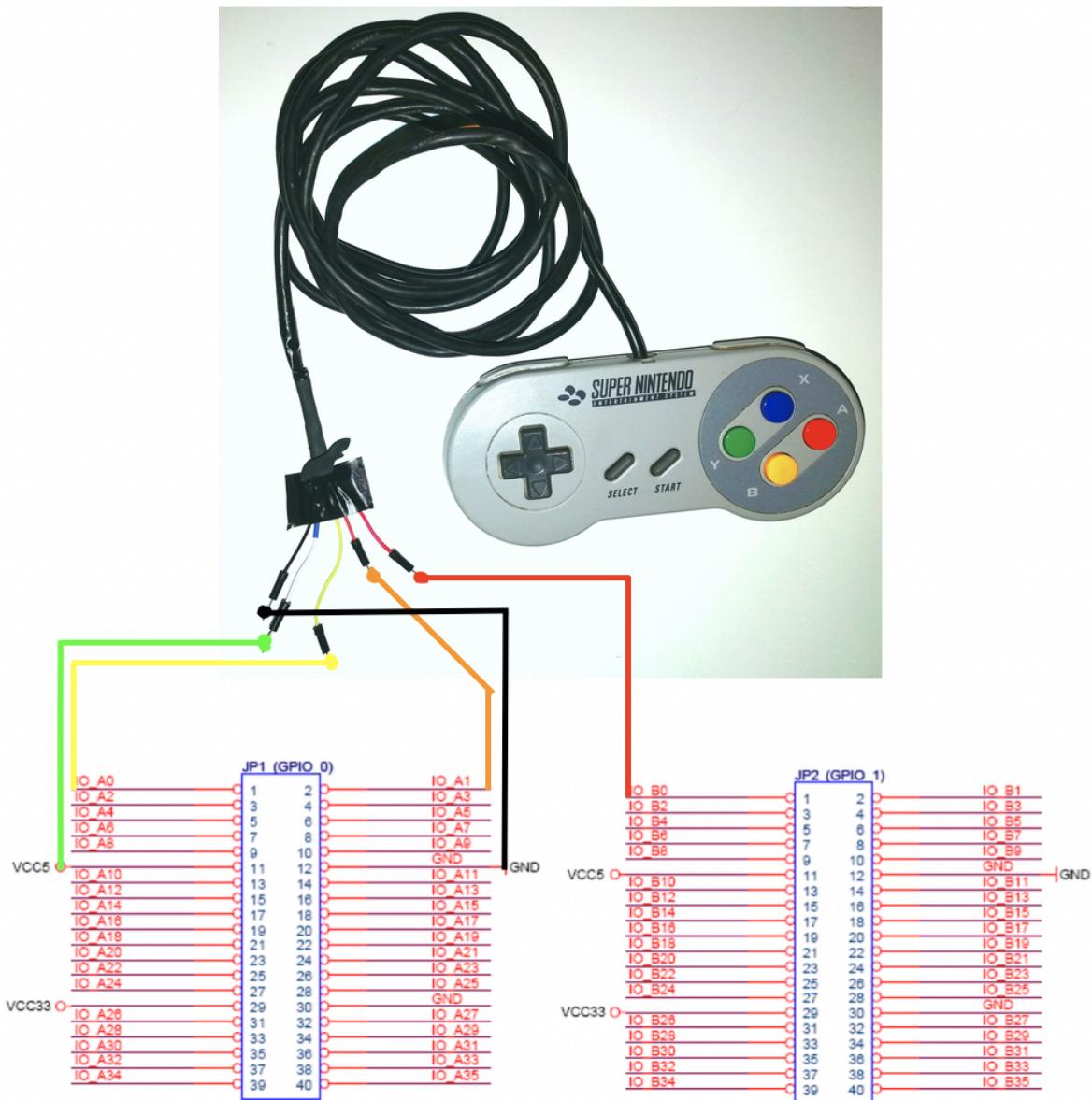
In particolare, all'attivazione del LATCH è rappresentato il campionamento della seguente combinazione di pulsanti: B, START, LEFT, X.



mario_snes

Il modulo *mario_snes* implementa il protocollo appena descritto interfacciandosi con il controller SNES tramite gli header di espansione a 40 pin GPIO_0 e GPIO_1 della DE1.

Il controller è così collegato ai GPIO:



Come si evince dalla Top-Level Entity, GPIO_0 e GPIO_1 sono configurati rispettivamente come output e input per la DE1.

Non tenendo in considerazione i pin di +5V e GND, al GPIO_0 sono collegati i seguenti pin di input per il controller: DATA CLOCK e DATA LATCH.

Al GPIO_1 invece è collegato il solo pin di output per il controller che è il SERIAL DATA.

L'entità *mario_snes* è organizzata come segue:

```

entity mario_snes is
  port (
    CLOCK      : in std_logic;
    RESET      : in std_logic;

    --PIN DI OUTPUT SNES
    CLOCK_PIN   : out std_logic;
    LATCH_PIN   : out std_logic;

    --PIN DI INPUT SNES
    DATA_PIN    : in std_logic;

    --SEGNALI INVIATI ALLA CONTROL UNIT

    LEFT_PRESSED : out std_logic;
    RIGHT_PRESSED : out std_logic;
    UP_PRESSED   : out std_logic;
    DOWN_PRESSED : out std_logic;

    B_PRESSED   : out std_logic;
    Y_PRESSED   : out std_logic;
    X_PRESSED   : out std_logic;
    A_PRESSED   : out std_logic;

    START_PRESSED : out std_logic;
    SELECT_PRESSED : out std_logic;

    L_PRESSED   : out std_logic;
    R_PRESSED   : out std_logic
  );
end entity;

```

All'interno del modulo vi è un unico process che implementa il protocollo SNES.
Il protocollo è semplicemente suddiviso in 4 fasi così organizzate:

1. LATCH

In questa fase si pone ad un livello logico 1 il segnale LATCH_PIN per un tempo pari a $12\mu s$, ovvero per un periodo pari a 600 attivazioni del clock a 50MHz.
Al raggiungimento dei $12\mu s$, il segnale LATCH_PIN transita al livello logico 0 e si entra nella fase WAIT CLOCK.

2. WAIT CLOCK

In questa fase si attende per un intervallo di $6\mu s$ (300 clock a 50MHz) allo scadere del quale viene portato ad un livello logico 0 il segnale CLOCK_PIN (solitamente a livello logico 1). Al termine si entra nella fase CLOCK.

3. CLOCK

Da questo momento in poi vengono inviati al controller 16 impulsi di CLOCK_PIN con periodo $12\mu s$ e duty cycle 50%.

Ad ogni fronte di discesa del CLOCK_PIN vengono effettuate le letture sullo stato di attivazione del segnale SERIAL_DATA proveniente dal controller.

Nel caso in cui il segnale SERIAL_DATA è ad un livello logico 0 si stabilisce l'attivazione dei segnali associati all'i-esimo periodo di CLOCK_PIN.

Al termine del 16-esimo periodo di CLOCK_PIN, si rispristina il valore logico 1 sul segnale CLOCK_PIN e si entra nella fase WAIT LATCH.

4. WAIT LATCH

In questa fase si attende per un intervallo di $6\mu s$ (300 clock a 50MHz) allo scadere del quale si ritorna alla fase LATCH.

Control Unit

La *Control Unit* si occupa principalmente di gestire l'avanzamento di stato del sistema e, tramite opportuni segnali di controllo, regola le attività svolte da *Datapath* e *View*. L'entità è strutturata come segue:

```
entity control_unit is
  port (
    CLOCK      : in std_logic;
    RESET      : in std_logic;

    --SEGNALI RICEVUTI DAL CONTROLLER SNES
    LEFT_PRESSED : in std_logic;
    RIGHT_PRESSED : in std_logic;
    UP_PRESSED : in std_logic;
    DOWN_PRESSED : in std_logic;
    B_PRESSED : in std_logic;
    Y_PRESSED : in std_logic;
    X_PRESSED : in std_logic;
    A_PRESSED : in std_logic;
    START_PRESSED : in std_logic;
    SELECT_PRESSED : in std_logic; --PAUSE
    L_PRESSED : in std_logic;
    R_PRESSED : in std_logic;

    --SEGNALI INVIATI AL DATAPATH
    SX : out std_logic;
    DX : out std_logic;
    UP : out std_logic;
    DOWN : out std_logic;
    B : out std_logic;
    Y : out std_logic;
    X : out std_logic;
    A : out std_logic;

    STATE : out mario_state;
    LEV_LOST: out std_logic;
    CURRENT_LEVEL: out integer;
    NEW_COIN : out std_logic;
    END_TIME : out std_logic;
    GAME_CLOCK : out std_logic;

    --SEGNALI RICEVUTI DAL DATAPATH
    LEVEL_LOADED : in std_logic;
    LIFE_LOST: in std_logic;
    LAST_LIFE_LOST : in std_logic;
    LEVEL_COMPLETE: in std_logic;
    COIN_CATCHED : in std_logic;
  );
end entity;
```

In questo modulo sono presenti 4 process così organizzati:

1. *ActionProcess*

Riceve dal modulo *mario_snes* la decodifica dei pulsanti premuti e li propaga al *Datapath* che li traduce in azioni su Mario;

2. *StateProcess*

Implementa il diagramma degli stati dell'intero sistema;

3. *GameTimingProcess*

Scandidisce i tempi di gioco relativi al completamento livello e timer monete;

4. *GameClockProcess*

Regola i tempi di elaborazione per il movimento di Mario.

ActionProcess e GameClockProcess

I process *ActionProcess* e *GameClockProcess* svolgono azioni molto semplici. Come già accennato, *ActionProcess* inoltra le richieste di movimento implementate dal modulo *mario_snes* ed è realizzato come segue:

```
ActionProces : process(CLOCK,RESET)
begin
    if (RESET='1') then
        SX <= '0';
        DX <= '0';
        UP <= '0';
        DOWN <= '0';

        B <= '0';
        Y <= '0';
        X <= '0';
        A <= '0';

    elsif rising_edge(CLOCK) then
        if (currState=PLAY) then
            SX <= LEFT_PRESSED;
            DX <= RIGHT_PRESSED;
            UP <= UP_PRESSED;
            DOWN <= DOWN_PRESSED;

            B <= B_PRESSED;
            Y <= Y_PRESSED;
            X <= X_PRESSED;
            A <= A_PRESSED;
        else
            SX <= '0';
            DX <= '0';
            UP <= '0';
            DOWN <= '0';

            B <= '0';
            Y <= '0';
            X <= '0';
            A <= '0';
        end if;
    end if;
end process;
```

GameClockProcess invece scandisce un tempo che regola l'elaborazione dei dati effettuata dal *Datapath*, questo per rallentare l'aggiornamento degli elementi visivi sullo schermo.

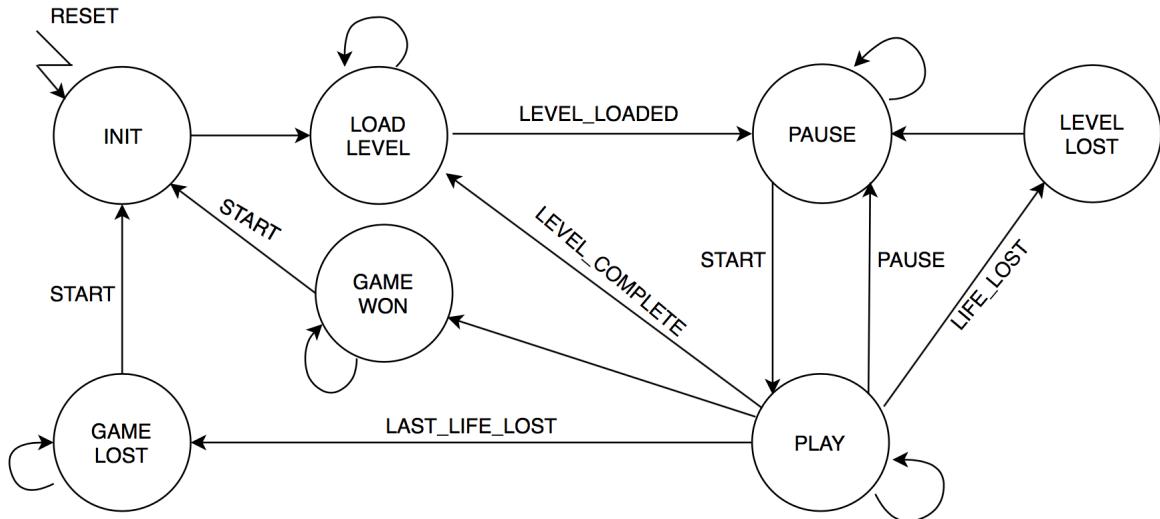
Senza tale processo ci siamo accorti che elaborando e presentando i dati al modulo VGA con una frequenza di 50MHz era impossibile seguire lo spostamento di Mario.

Di seguito la semplice realizzazione del *GameClockProcess* che attiva il segnale *GAME_CLOCK* al raggiungimento della soglia *GAME_CLOCK_PRESCALER*:

```
GameClockProcess : process(CLOCK, RESET)
begin
    if (RESET = '1') then
        game_clock_counter <= 0;
        GAME_CLOCK <= '0';
    elsif (rising_edge(clock)) then
        if(game_clock_counter < GAME_CLOCK_PRESCALER-1) then
            game_clock_counter <= game_clock_counter+1;
            GAME_CLOCK <= '0';
        else
            game_clock_counter <= 0;
            GAME_CLOCK <= '1';
        end if;
    end if;
end process;
```

StateProcess

L'intero sistema è regolato dal process *StateProcess* che implementa una macchina a stati di Moore definita come segue:



Il process riceve i segnali di input dal *Datapath* e dal modulo *mario_snes*, eventualmente provvede a modificare lo stato del sistema che viene poi propagato a tutte le entità affinché possano reagire correttamente.

I possibili stati del sistema sono rappresentati nel seguente modo e sono così definiti:

```
--STATI DEL SISTEMA
type mario_state is (INIT, PAUSE, PLAY, LOAD_LEVEL, LEVEL_LOST, GAMELOST, GAMEWON);
```

- **INIT**
È lo stato iniziale del sistema e si occupa di ripristinare il livello logico del segnale di controllo LEV_LOST e del valore del segnale CURRENT_LEVEL.
È uno stato di transizione in quanto si permane per un solo periodo di clock prima di transitare nello stato LOAD LEVEL;
Si giunge in questo stato ad ogni attivazione del segnale RESET e ad ogni attivazione del segnale START (da controller SNES) nel caso in cui lo stato corrente sia GAME LOST oppure GAME WON.
- **LOAD LEVEL**
In questo stato si attende da parte del *Datapath* l'attivazione del segnale LEVEL_LOADED che notifica l'avvenuto caricamento in SRAM del livello indicato da CURRENT_LEVEL.
La ricezione del segnale LEVEL_LOADED implica la transizione nello stato PAUSE
- **PAUSE**
È lo stato di pausa del gioco in cui il sistema è fermo; Non viene effettuata alcuna operazione fino alla ricezione del segnale START attivato dal giocatore tramite il controller SNES che implica la transizione nello stato PLAY e l'inizio del gioco.

- **PLAY**

Il sistema è avviato e ad ogni clock le entità elaborano i segnali e si scambiano informazioni. Vengono inoltrati al *Datapath* i segnali per lo spostamento di Mario e vengono effettuati gli appositi controlli riguardo eventuali collisioni di Mario con mattoncini e monete.

La transizione verso un altro stato della macchina è dovuta alla ricezione di uno dei seguenti segnali provenienti dal *Datapath* o da una richiesta esplicita del giocatore:

- **PAUSE**

Segnale attivato dal giocatore tramite il pulsante SELECT sul controller SNES che porta il sistema nell'omonimo stato PAUSE

- **LEVEL COMPLETE**

Segnale inviato dal *Datapath* al completamento del livello, ovvero quando è stato catturato il numero previsto di monete.

Implica una transizione nello stato di CHANGELEVEL nel caso in cui CURRENT_LEVEL non abbia raggiunto il numero massimo di livelli contenuti nel sistema, altrimenti si transita automaticamente nello stato di GAMEWON

- **LIFE LOST**

Segnale inviato dal *Datapath* nel caso in cui il giocatore non raggiunge l'obiettivo entro il tempo di gioco previsto. La ricezione di questo segnale implica la transizione nello stato LEVEL LOST e l'attivazione del segnale di controllo LEV_LOST inviato alla *View*

- **LAST LIFE LOST**

Segnale inviato dal *Datapath* nel caso in cui il giocatore non raggiunge l'obiettivo entro il tempo di gioco previsto e ha perso l'ultima vita.

La ricezione di questo segnale implica la transizione nello stato GAME LOST

- **LEVEL LOST**

Come INIT anche in questo caso si permane per un solo periodo di clock prima di transitare nello stato di PAUSE per rigiocare il livello.

- **GAMELOST**

È lo stato di perdita del gioco, si attende che il giocatore prema il pulsante START sul controller SNES per poi transitare nello stato INIT.

- **GAMEWON**

È lo stato di vittoria del gioco, si attende che il giocatore prema il pulsante START sul controller SNES per poi transitare nello stato INIT.

Di seguito l'implementazione del diagramma degli stati appena descritto:

```

StateProcess : process(CLOCK, RESET)
begin
    if (RESET='1') then
        currState<=INIT;
    elsif (rising_edge(CLOCK)) then
        case currState is
            --INIT
            when INIT=>
                currentLevel<=0;
                currState <= LOAD_LEVEL;
                LEV_LOST<='0';
            --LOAD_LEVEL
            when LOAD_LEVEL=>
                if(LEVEL_LOADED='1') then
                    currState <= PAUSE;
                end if;
            --PAUSE
            when PAUSE=>
                if (START_PRESSED = '1' ) then
                    currState <= PLAY;
                    LEV_LOST<='0';
                end if;
            --PLAY
            when PLAY=>
                if (SELECT_PRESSED = '1') then
                    currState <= PAUSE;
                elsif(LEVEL_COMPLETE='1') then
                    if(currentLevel<NUM_LEVELS-1) then
                        currentLevel<=currentLevel+1;
                        currState <= LOAD_LEVEL;
                    else
                        currState <= GAMEWON;
                    end if;
                elsif(LIFE_LOST='1') then
                    currState <= LEVEL_LOST;
                    LEV_LOST<='1';
                elsif (LAST_LIFE_LOST='1') then
                    currState <= GAMELOST;
                end if;
            --LEVEL_LOST
            when LEVEL_LOST=>
                currState <= PAUSE;
            --GAMELOST
            when GAMELOST=>
                if (START_PRESSED='1') then
                    currState <= INIT;
                end if;
            --GAMEWON
            when GAMEWON=>
                if (START_PRESSED='1' ) then
                    currState <= INIT;
                end if;
        end case;
    end if;
end process;

```

GameTimingProcess

Il suo compito è quello di scandire i tempi di gioco relativi al completamento del livello e al timer associato alle monete. Il giocatore ha un rapporto diretto con due tempi:

1. Timer di gioco

Il tempo a disposizione per il completamento di ciascun livello è espresso in secondi ed è rappresentato dalla costante SECONDS_PER_LEVEL il cui valore massimo è 99.

Durante lo stato PLAY viene mostrato al giocatore un countdown che lo tiene informato sul tempo residuo per raggiungere l'obiettivo.

Allo scadere del tempo si pone al livello logico 1, per una durata di un solo periodo di clock, un segnale denominato END_TIME che notifica al *Datapath* lo scadere del tempo. In questo caso il *Datapath* risponde attivando il segnale LIFE LOST o LAST LIFE LOST sopra descritti.

2. Timer moneta

Il tempo massimo di permanenza di una moneta in una certa posizione nello schermo è stabilita dalla costante TIMER_COIN fissata a 5 secondi.

La difficoltà introdotta al giocatore è definita da questo timer che varia in base al livello di gioco corrente.

Al crescere del livello il timer moneta diminuisce di 1 secondo.

Allo scadere del timer viene posto al livello logico 1 il segnale NEW_COIN inviato al *Datapath* che provvede a rimuovere la moneta dallo schermo e a sostituirla con un'altra.

Allo stesso modo alla ricezione del segnale COIN_CATCHED proveniente dal *Datapath* che notifica l'avvenuta cattura della moneta, il process risponde ponendo al livello logico 1 il segnale NEW_COIN.

```
case currState is
  when PLAY =>
    NEW_COIN<='0';
    END_TIME <='0';
    --NEW_COIN_TIMER
    if (stop=false) then
      coin_timer<=TIMER_COIN-(currentLevel*ONE_SEC);
      if (new_coin_timer<coin_timer) then
        new_coin_timer <= new_coin_timer+1;
      else
        new_coin_timer<=0;
        NEW_COIN<='1';
      end if;
    --COIN_CATCHED
    if(COIN_CATCHED='1')then
      new_coin_timer<=0;
      NEW_COIN<='1';
    end if;
    --END_TIME_TIMER
    if (one_sec_timer<ONE_SEC) then
      one_sec_timer <= one_sec_timer+1;
    else
      one_sec_timer<=0;
      if(end_time_timer>0) then
        end_time_timer<= end_time_timer-1;
        END_TIME <='0';
      else
        end_time_timer<=SECONDS_PER_LEVEL;
        END_TIME <='1';
        NEW_COIN<='0';
        stop:=true;
      end if;
    end if;
  end if;
when PAUSE =>
```

Datapath

Ai fini dell'elaborazione inerenti alle possibili collisioni, si è scelto di mappare lo scenario in SRAM perché si è ritenuto più semplice effettuare le opportune operazioni di controllo legate allo spostamento di Mario nella schermata di gioco.

In SRAM vi è un'esatta copia dello scenario presentato a video, quindi un'associazione diretta tra coordinata video e indirizzo di memoria.

Ad ogni movimento effettuato da Mario si stabilisce quindi il rapporto causa-effetto.

Si è scelto di rappresentare Mario nel sistema tramite una coppia di coordinate (X, Y) che identificano il punto superiore sinistro dello sprite avente larghezza e altezza definite dalle rispettive costanti MARIOWIDTH e MARIOHEIGHT.

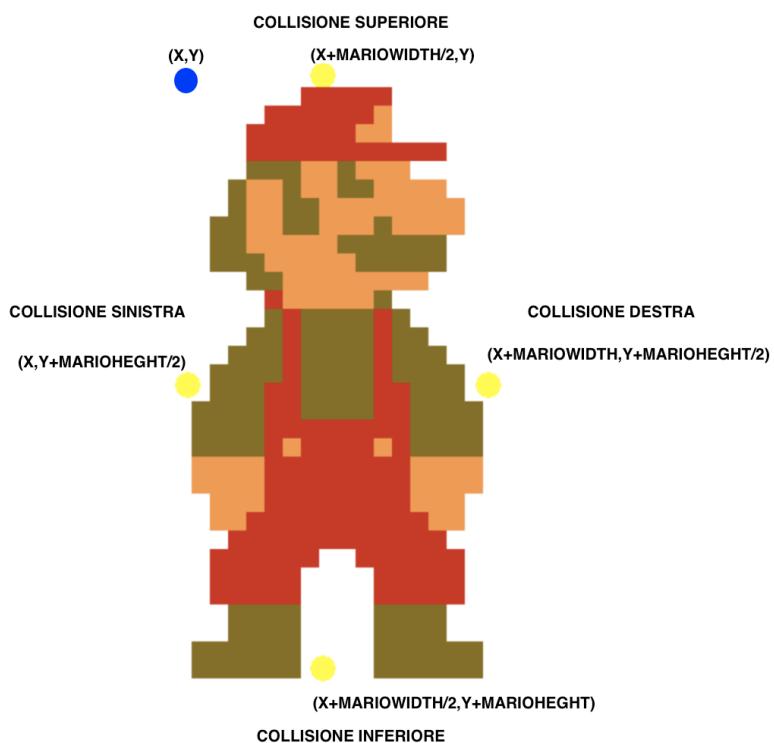
Il sistema di riferimento coincide con quello usato dallo standard VGA in cui l'origine viene individuata nel lato superiore sinistro dello schermo.

Come accennato è stata fatta un'associazione diretta tra coordinate e indirizzi SRAM tali per cui ad ogni coordinata sullo schermo è associato un indirizzo di memoria.

Quando si vuole controllare se Mario può spostarsi in una certa direzione, si trova il punto in cui ci si vuole spostare a partire dalla posizione corrente.

Dopo aver trovato il punto di destinazione, si legge al corrispondente indirizzo della SRAM il tipo di elemento rappresentato. Se tale punto rappresenta un elemento impenetrabile, allora viene rilevata una collisione e la richiesta di movimento viene annullata, in caso contrario viene consentita.

In base alla direzione in cui è stato richiesto lo spostamento, sono localizzati diversi punti cardine tramite i quali vengono effettuate le opportune valutazioni.

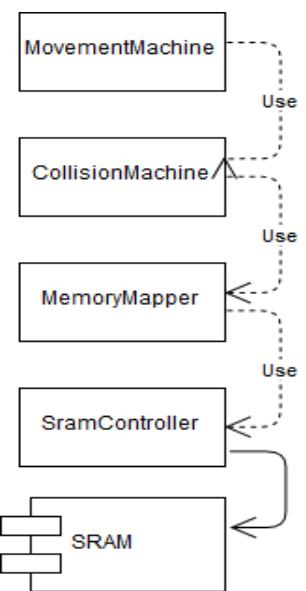


Il sistema di movimento si basa su quattro componenti principali, essi interagiscono in modo tale che ognuno utilizza il successivo con un modello quasi black-box.

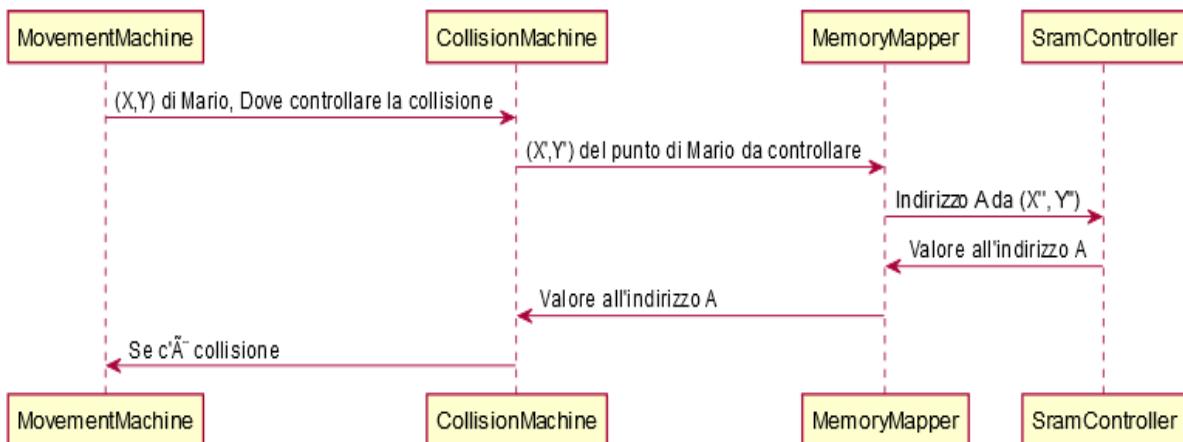
Ciò rende l'intera architettura più scalabile.

L'elemento top level è la *MovementMachine* che fornisce la coordinata X e Y corrente di Mario all'intero sistema. Le richieste di movimento ricevute dalla *Control Unit* vengono elaborate dalla *MovementMachine* che con il supporto della *CollisionMachine* definisce se Mario può muoversi in quella direzione oppure no. Per verificare tale richiesta, la *CollisionMachine* si interfaccia con il *MemoryMapper* che a sua volta, con il supporto dell'*SRAMController* gestisce lo scambio dati con la memoria.

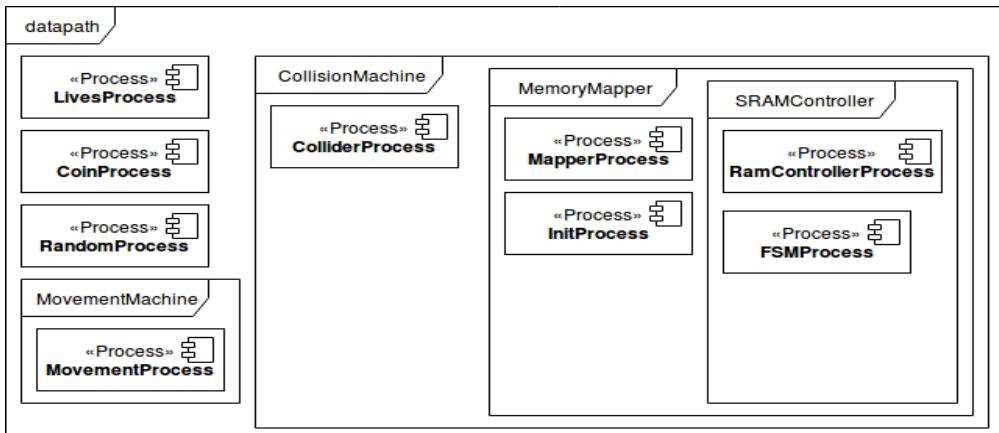
Il *MemoryMapper* effettua anche tutte le operazioni necessarie per decodificare un eventuale collisione a partire da una coppia di coordinate.



Per maggiore chiarezza è riportato il diagramma di sequenza delle richieste effettuate dalla *MovementMachine* alle sue sotto unità e i parametri scambiati:



Il *Datapath* è costituito da un insieme di moduli di seguito riportati:



- *mario_datapath*

Implementa i process incaricati alla gestione delle vite, alla rappresentazione e cattura delle monete. I due process sono rispettivamente *LivesProcess* e *CoinProcess*. Inoltre, istanzia le entità preposte alla gestione del movimento e alla verifica delle collisioni sfruttando il mapping in SRAM.

- *movement_machine*

Implementa un unico process che ha il compito di memorizzare e aggiornare la coppia di coordinate di Mario a fronte di una richiesta di movimento proveniente dalla *Control Unit*.

- *collision_machine*

Istanzia al suo interno il *MemoryMapper* (contenuto nel modulo *memory_mapper*) tramite il quale stabilisce una eventuale collisione con un elemento a partire dalle coordinate ricevute dal modulo *MovementMachine*.

- *memory_mapper*

Inizializza e regola le scritture in SRAM interfacciandosi con l'*SRAMController* istanziato al suo interno

- *sram_controller*

Si occupa di regolare gli accessi alla memoria SRAM.

mario_datapath

Come già accennato, questo modulo implementa i seguenti process così organizzati:

- *LivesProcess*

Il processo *LivesProcess* si occupa di effettuare le operazioni necessarie alla gestione delle vite di Mario.

Nello specifico, pilota due segnali di uscita denominati LIFE_LOST e LAST_LIFE_LOST i quali hanno un valore logico di default imposto a 0 e vengono portati al livello logico 1 per un solo periodo di clock al verificarsi delle rispettive condizioni già descritte (vedi *StateProcess*).

Se lo stato del sistema è INIT, viene ripristinato il numero di vite correnti ad un valore di default imposto da DEFAULT_NUM_LIVES, altrimenti l'elaborazione è regolata dalla ricezione del segnale END_TIME che viene generato dalla *Control Unit* solo nello stato PLAY.

Il segnale num_lives è un counter che tiene aggiornato il numero di vite corrente ed è propagato ad ogni periodo di clock sul segnale di uscita LIVES.

Di seguito la realizzazione del process descritto:

```
LivesProcess : process(CLOCK, RESET)
begin
    if(RESET = '1') then
        num_lives <= DEFAULT_NUM_LIVES;
    elsif(rising_edge(CLOCK)) then
        LAST_LIFE_LOST     <= '0';
        LIFE_LOST         <= '0';

        if (STATE=INIT) then
            num_lives <= DEFAULT_NUM_LIVES;
        elsif(END_TIME = '1') then
            if(s_num_coin_catched < NUM_COIN_PER_LEVEL) then
                num_lives <= num_lives - 1;
                if(num_lives - 1 = 0) then
                    LAST_LIFE_LOST <= '1';
                else
                    LIFE_LOST <= '1';
                end if;
            end if;
        else
            if (s_num_coin_catched = NUM_COIN_PER_LEVEL AND num_lives<DEFAULT_NUM_LIVES) then
                num_lives <= num_lives + 1;
            end if;
        end if;
    end if;
end process LivesProcess;
```

- *CoinProcess*

Il processo *CoinProcess* si occupa di presentare alla *View* le coordinate della moneta e di rilevarne la cattura.

Si occupa ulteriormente di tenere il conto delle monete attualmente catturate e di generare il segnale *LEVEL_COMPLETE* al raggiungimento dell’obiettivo.

Le monete sono raggruppate per livelli e sono memorizzate in una struttura dati del tipo *matrix_coins* così definito:

```
type t_coins_array is array(0 to MAX_COIN-1) of coord; -- coordinate type
type matrix_coins is array (0 to NUM_LEVELS-1) of t_coins_array;

constant LEVEL_C1 : t_coins_array := (C_11, C_12, C_13, C_14, C_15, C_16, C_17, C_18, C_19, C_110);
constant LEVEL_C2 : t_coins_array := (C_21, C_22, C_23, C_24, C_25, C_26, C_27, C_28, C_29, C_210);
constant LEVEL_C3 : t_coins_array := (C_31, C_32, C_33, C_34, C_35, C_36, C_37, C_38, C_39, C_310);
constant LEVEL_C4 : t_coins_array := (C_41, C_42, C_43, C_44, C_45, C_46, C_47, C_48, C_49, C_410);
constant LEVEL_C5 : t_coins_array := (C_51, C_52, C_53, C_54, C_55, C_56, C_57, C_58, C_59, C_510);

constant GAME_COINS : matrix_coins := (LEVEL_C1, LEVEL_C2, LEVEL_C3, LEVEL_C4, LEVEL_C5);
```

La selezione della moneta è affidata ad un process denominato *RandomProcess* che definisce in modo casuale l’i-esimo elemento da estrarre dalle strutture dati associate a ciascun livello (LEVEL_C1, LEVEL_C2...)

Ogni elemento della struttura dati rappresenta una moneta secondo una coppia di coordinate X e Y che ne identificano il punto superiore sinistro e la rispettiva posizione nello schermo.

Alla ricezione del segnale *NEW_COIN* vengono aggiornate le coordinate della moneta relative all’i-esimo elemento definito da *RandomProcess*.

Alla ricezione del segnale *END_TIME* proveniente dalla *Control Unit* si effettua il reset del contatore interno relativo al numero di monete catturate.

La cattura delle monete è realizzata effettuando un controllo circoscritto alla coordinata della moneta attualmente rappresentata.

```
if(s_coin_catched = '0' and(
    ((s_mario_x + MARIO_WIDTH) = s_coin_x and s_mario_y < (s_coin_y + COIN_HEIGHT) and (s_mario_y + MARIO_HEIGHT) > s_coin_y) or
    (s_mario_x = (s_coin_x + COIN_WIDTH) and s_mario_y < (s_coin_y + COIN_HEIGHT) and (s_mario_y + MARIO_HEIGHT) > s_coin_y) or
    ((s_mario_y + MARIO_HEIGHT) = s_coin_y and s_mario_x < (s_coin_x + COIN_WIDTH) and (s_mario_x + MARIO_WIDTH) > s_coin_x) or
    (s_mario_y = (s_coin_y + COIN_HEIGHT) and s_mario_x < (s_coin_x + COIN_WIDTH) and (s_mario_x + MARIO_WIDTH) > s_coin_x)
)
) then
    s_coin_catched <= '1';
    s_num_coin_catched <= s_num_coin_catched + 1;
end if;
```

Se la coppia di coordinate che rappresenta la posizione di Mario è interna all’area della moneta allora questa si considera catturata e viene emesso il segnale *COIN_CATCHED* che viene elaborato dalla *Control Unit*.

Se lo stato di gioco è *PLAY* e il numero di monete è pari all’obiettivo definito dalla costante *NUM_COIN_PER_LEVEL* allora viene asserita l’uscita *LEVEL_COMPLETE* che informa la *Control Unit* del completamento del livello.

movement_machine

Questa entità si occupa di muovere Mario all'interno dell'ambiente creato in ottemperanza alle restrizioni imposte dalla *CollisionMachine*.

L'entità è organizzata come segue:

```
entity movement_machine is
  port (
    CLOCK : in std_logic; -- clock in
    RESET : in std_logic; -- reset async

    I_CONTROLLER_BUTTON_SX : in std_logic; -- controller sx button
    I_CONTROLLER_BUTTON_DX : in std_logic; -- controller dx button
    I_CONTROLLER_BUTTON_UP : in std_logic; -- controller up button
    I_COLLISION_SX : in std_logic; -- collision sx
    I_COLLISION_DX : in std_logic; -- collision dx
    I_COLLISION_UP : in std_logic;
    I_COLLISION_DOWN : in std_logic; -- collision down
    I_GAME_LOGIC_CLOCK : in std_logic; -- clockish

    D_LED : out std_logic_vector(3 downto 0);

    O_REQUEST_COLLISION_SX : out std_logic; -- request to move sx
    O_REQUEST_COLLISION_DX : out std_logic; -- request to move dx
    O_REQUEST_COLLISION_DOWN : out std_logic; -- request to move down
    O_REQUEST_COLLISION_UP : out std_logic; -- request to move up
    O_REQUEST_NEXT_X : out natural; -- x of the request
    O_REQUEST_NEXT_Y : out natural; -- y of the request
    O_X : out natural; -- actual next x
    O_Y : out natural -- actual next y
  );
end entity;
```

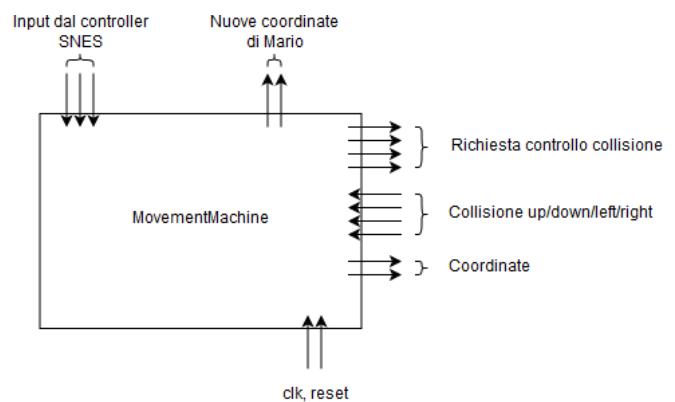
Questo modulo riceve in ingresso i segnali provenienti dalla *Control Unit* tramite i quali calcola la prossima posizione fornita che viene fornita al resto del sistema tramite le uscite O_X e O_Y.

Dato che si interfaccia con moduli che utilizzano il controllore SRAM, è costretta ad effettuare un solo controllo per ogni ciclo di clock. Ciò significa che non è possibile controllare due punti di collisione nello stesso periodo di clock. Per questo motivo si è scelto di valutare in modo alternato le possibili collisioni. In un primo ciclo di clock vengono valutate le collisioni orizzontali limitate al solo punto di riferimento associato alla direzione di movimento e durante il ciclo di clock successivo quelle verticali, anch'esse limitate al punto di riferimento associato alla direzione di movimento.

Ad ogni verifica di collisione è associato un segnale che esplicita al modulo *collision_machine* il tipo di collisione che deve controllare.

Di seguito i segnali riportati:

- O_REQUEST_COLLISION_SX
- O_REQUEST_COLLISION_DX
- O_REQUEST_COLLISION_DOWN
- O_REQUEST_COLLISION_UP



Ulteriormente vengono inoltrati al modulo *collision_machine* le coordinate da verificare rappresentate da:

- O_REQUEST_NEXT_X
- O_REQUEST_NEXT_Y

Una volta inoltrata la richiesta al modulo *collision_machine* si attende l'esito ricevuto tramite i seguenti segnali:

- I_COLLISION_SX
- I_COLLISION_DX
- I_COLLISION_UP
- I_COLLISION_DOWN

Questo modulo descrive anche la traiettoria di un eventuale salto calcolando il punto più alto da raggiungere in base alla costante MAX_JUMP_HEIGHT e in seguito, decrementando la coordinata Y a meno di una collisione verticale.

Per descrivere una traiettoria che approssima una parabola durante l'azione di salto, è stato deciso di introdurre dei cicli di *waste* all'interno dei quali non vengono inoltrate richieste di collisione al modulo *collision_machine*.

Tali cicli di *waste* sono effettuati sia per gli spostamenti in verticale che per quelli in orizzontale, rispettivamente modellati per ottenere la traiettoria voluta.

Per imporre una diversa traiettoria è sufficiente modificare le seguenti costanti che regolano i cicli di *waste* associati:

1. CYCLES_TO_WASTE_JUMP
2. CYCLES_TO_WASTE_MOVEMENT

Un ulteriore rallentamento del sistema è garantito dal segnale ricevuto dalla *Control Unit I_GAME_CLOCK*.

```

if S_PROCESS_JUMP_PHASE then
    if S_WASTE_CYCLES_JUMP = 0 then
        D_LED(2) <= I_CONTROLLER_BUTTON_UP;

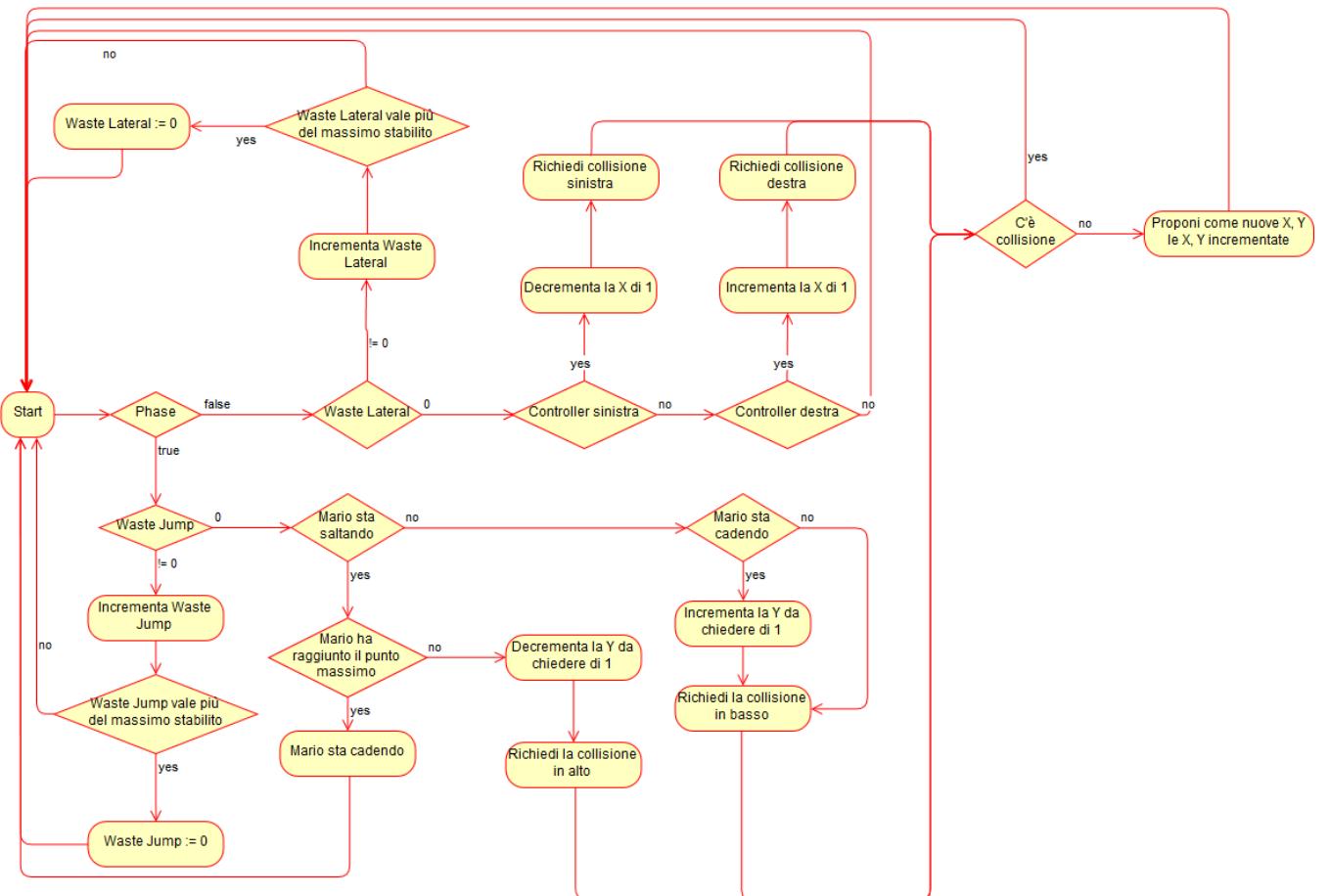
        O_REQUEST_COLLISION_DOWN <= '0';
        if I_CONTROLLER_BUTTON_UP = '1' and not IS_JUMPING and not IS_FALLING then
            IS_JUMPING := true;
            S_JUMP_HIGHEST_POINT <= NEW_Y - MAX_JUMP_HEIGHT;
        end if;

        if IS_JUMPING then
            if NEW_Y <= S_JUMP_HIGHEST_POINT then -- reached vertex
                IS_JUMPING := false;
                IS_FALLING := true;
            else
                O_REQUEST_COLLISION_UP <= '1'; -- requests if we can go up
                NEW_Y := NEW_Y - 1;
            end if;
        elsif IS_FALLING then -- we are falling! (and can't get up! D:)
            O_REQUEST_COLLISION_DOWN <= '1';
            NEW_Y := NEW_Y + 1;
        else
            O_REQUEST_COLLISION_DOWN <= '1';
        end if;
        O_REQUEST_NEXT_X <= NEW_X; -- send the request
        O_REQUEST_NEXT_Y <= NEW_Y;

        D_LED(1) <= I_COLLISION_DOWN;
        if I_COLLISION_DOWN = '1' then
            if IS_FALLING then
                NEW_Y := NEW_Y - 1;
                IS_FALLING := false;
            end if;
        else
            IS_FALLING := true;
        end if;
        if I_COLLISION_UP = '1' and IS_JUMPING then
            NEW_Y := NEW_Y + 1;
            IS_JUMPING := false;
            IS_FALLING := true;
        end if;
    end if;
    S_WASTE_CYCLES_JUMP <= S_WASTE_CYCLES_JUMP + 1 rem CYCLES_TO_WASTE_JUMP; -- waste clocks

```

Di seguito è riportato il diagramma di funzionamento del sistema appena descritto. Si notino i cicli di *waste* per rallentare l'esecuzione del gioco e il sistema di richiesta riguardo la presenza o meno della collisione:



collision_machine

La *CollisionMachine* è l'entità che si occupa di verificare se Mario collide o meno con uno dei blocchi, col pavimento o con i bordi dello schermo ed è organizzata come segue:

```

entity collision_machine is
  port (
    CLOCK : in std_logic; -- clock in
    RESET : in std_logic; -- reset async

    I_X_TO_CHECK : in natural; -- x coordinate
    I_Y_TO_CHECK : in natural; -- y coordinate
    I_CE : in std_logic; -- chip enable

    I_REQUEST_LEFT_COLLISION : in std_logic;
    I_REQUEST_RIGHT_COLLISION : in std_logic;
    I_REQUEST_BOTTOM_COLLISION : in std_logic;
    I_REQUEST_TOP_COLLISION : in std_logic;

    I_CURRENT_LEVEL_NUMBER : in integer;
    I_STATE : in mario_state;

    O_LEVEL_LOADED_DONE : out std_logic;

    O_COLLISION_LEFT : out std_logic; -- left collision
    O_COLLISION_RIGHT : out std_logic; -- right collision
    O_COLLISION_TOP : out std_logic; -- top collision
    O_COLLISION_BOTTOM : out std_logic; -- bottom collision

    D_OUT : out std_logic_vector(5 downto 0);

    -- ram mapping
    SRAM_ADDR : out std_logic_vector(17 downto 0);
    SRAM_DQ : inout std_logic_vector(15 downto 0);
    SRAM_CE_N : out std_logic;
    SRAM_OE_N : out std_logic;
    SRAM_WE_N : out std_logic;
    SRAM_UB_N : out std_logic;
    SRAM_LB_N : out std_logic
  );
end entity;

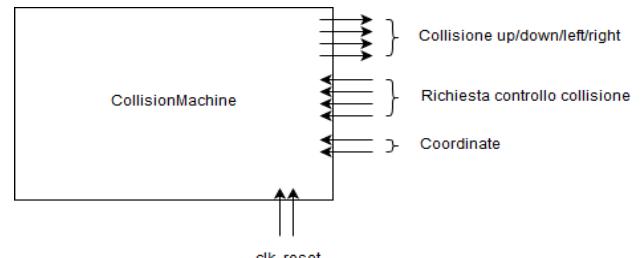
```

Questa entità si occupa solo di segnalare una eventuale collisione con un elemento.

Per verificare i punti di collisione, come già accennato, sono stati stabiliti dei punti cardine localizzati attorno allo sprite di Mario.

A ciascuna richiesta di verifica di collisione è associato un punto cardine che viene estratto a partire dalle coordinate

rappresentate rispettivamente da *I_X_TO_CHECK* e *I_Y_TO_CHECK*.



RICHIESTA	X	Y
REQUEST_LEFT_COLLISION	<i>I_X_TO_CHECK</i>	<i>I_Y_TO_CHECK+HALF_MARIO_HEIGHT</i>
<i>I_REQUEST_RIGHT_COLLISION</i>	<i>I_X_TO_CHECK+MARIOWIDTH</i>	<i>I_Y_TO_CHECK+HALF_MARIO_HEIGHT</i>
<i>I_REQUEST_BOTTOM_COLLISION</i>	<i>I_X_TO_CHECK+HALF_MARIO_WIDTH</i>	<i>I_Y_TO_CHECK+MARIOHEIGHT</i>
<i>I_REQUEST_TOP_COLLISION</i>	<i>I_X_TO_CHECK+HALF_MARIO_WIDTH</i>	<i>I_Y_TO_CHECK</i>

Dopo aver calcolato il punto cardine da verificare, questo viene inviato al modulo *memory_mapper* che ne restituisce il contenuto in memoria (16 bit) rappresentato da *S_DATA_FROM_MEMORY* sul quale vengono effettuati gli opportuni controlli per stabilire la collisione.

Di seguito il process che implementa le azioni appena descritte:

```

ColliderProcess : process(CLOCK, RESET)--, I_X_TO_CHECK, I_Y_TO_CHECK)
begin
    if RESET = '1' then -- reset procedure
        -- reset all outputs
        O_COLLISION_LEFT <= '0';
        O_COLLISION_RIGHT <= '0';
        O_COLLISION_BOTTOM <= '0';
        O_COLLISION_TOP <= '0';
        S_X_TO_CHECK <= I_X_TO_CHECK;
        S_Y_TO_CHECK <= I_Y_TO_CHECK;
    elsif rising_edge(CLOCK) then
        -- border detection
        -- reset all outputs
        O_COLLISION_LEFT <= '0';
        O_COLLISION_RIGHT <= '0';
        O_COLLISION_BOTTOM <= '0';
        O_COLLISION_TOP <= '0';
        if I_REQUEST_LEFT_COLLISION = '1' then
            S_X_TO_CHECK <= I_X_TO_CHECK;
            S_Y_TO_CHECK <= I_Y_TO_CHECK + HALF_MARIO_HEIGHT;
            if S_DATA_FROM_MEMORY(0) = '1' or I_X_TO_CHECK <= LEFT_BORDER then
                O_COLLISION_LEFT <= '1';
            end if;
        elsif I_REQUEST_RIGHT_COLLISION = '1' then
            S_X_TO_CHECK <= I_X_TO_CHECK + MARIOWIDTH;
            S_Y_TO_CHECK <= I_Y_TO_CHECK + HALF_MARIO_HEIGHT;
            if S_DATA_FROM_MEMORY(0) = '1' or I_X_TO_CHECK + MARIOWIDTH >= RIGTH_BORDER then
                O_COLLISION_RIGHT <= '1';
            end if;
        elsif I_REQUEST_BOTTOM_COLLISION = '1' then
            S_X_TO_CHECK <= I_X_TO_CHECK + HALF_MARIO_WIDTH;
            S_Y_TO_CHECK <= I_Y_TO_CHECK + MARIOHEIGHT;
            if S_DATA_FROM_MEMORY(0) = '1' or I_Y_TO_CHECK + MARIOHEIGHT >= BOTTOM_BORDER then
                O_COLLISION_BOTTOM <= '1';
            end if;
        elsif I_REQUEST_TOP_COLLISION = '1' then
            S_X_TO_CHECK <= I_X_TO_CHECK + HALF_MARIO_WIDTH;
            S_Y_TO_CHECK <= I_Y_TO_CHECK;
            if S_DATA_FROM_MEMORY(0) = '1' or I_Y_TO_CHECK <= TOP_BORDER then
                O_COLLISION_TOP <= '1';
            end if;
        end if;
        D_OUT <= S_DATA_FROM_MEMORY(2 downto 0) & I_REQUEST_LEFT_COLLISION & I_REQUEST_BOTTOM_COLLISION & I_REQUEST_RIGHT_COLLISION;
    end if;
end process;

```

memory_mapper

Il *MemoryMapper* si occupa di interfacciarsi con l'*SRAMController* per restituire, dato un punto (X, Y), il vettore memorizzato all'indirizzo corrispondente.

L'entità è organizzata come segue:

```
entity memory_mapper is
  port (
    CLOCK : in std_logic; -- clock in
    RESET : in std_logic; -- reset async

    I_X_TO_CHECK : in natural; -- x coordinate
    I_Y_TO_CHECK : in natural; -- y coordinate
    I_CE : in std_logic; -- chip enable

    I_CURRENT_LEVEL_NUMBER : in integer;
    I_STATE : in mario_state;

    O_LEVEL_LOADED_DONE : out std_logic;

    O_DATA_FROM_MEMORY : out std_logic_vector(15 downto 0); -- data out for the debug
    O_IS_UPDATING : out std_logic; -- if the memory is resetting;

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

Dato che lo spazio in SRAM non era sufficiente per memorizzare l'intero scenario, si è scelto di parametrizzare la rappresentazione riducendo di $\frac{1}{2}$ lo spazio da utilizzare. In particolare, per avere una rappresentazione ridotta rispetto alle coordinate ricevute, si considerano le coordinate in valore dimezzato: $\left(\frac{x}{2}, \frac{y}{2}\right)$

A queste coordinate viene applicato il seguente algoritmo di conversione in indirizzi SRAM:

$$\text{Address} = \frac{y}{2} * 240 + \frac{x}{2}$$

Questa operazione permette di mappare una matrice concettualmente bidimensionale su un vettore monodimensionale e ottenere una compressione maggiore.

Il modulo si occupa anche di caricare in RAM i livelli quando si è nello stato LOAD_LEVEL specificato dalla *Control Unit*.

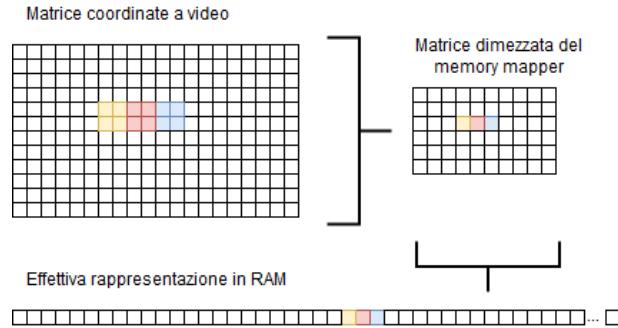
Prima di effettuare delle operazioni di scrittura viene eseguita una operazione di reset di tutto il contenuto della SRAM. Durante tali operazioni è asserito il segnale O_IS_UPDATING il quale informa i moduli che lo usano che non può soddisfare le richieste. Al termine dell'operazione di caricamento del livello viene inviata una conferma alla *Control Unit* tramite il segnale O_LEVEL_LOADED_DONE.

Il modulo ordina le scritture in RAM e di conseguenza definisce la rappresentazione in memoria dei blocchi.

Un blocco è un vettore di 16 bit avente il bit meno significativo pari a 1, questo permette, in caso di implementazioni o miglioramenti futuri, una maggiore scalabilità.

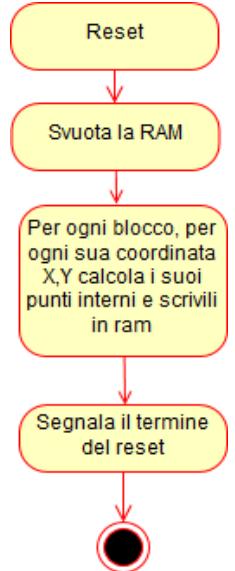


Lo schema di conversione delle coordinate video in indirizzi SRAM è il seguente:



Di seguito sono riportati rispettivamente il diagramma che descrive l'operazione di reset della SRAM e il process *MapperProcess* che lo implementa:

```
MapperProcess : process(CLOCK, RESET)
begin
if rising_edge(CLOCK) then
    O_LEVEL_LOADED_DONE <= '0';
    if S_RESET then -- soft reset in place
        if S_RESET_ADDR < MAX_RAM_REPRESENTATION_HEIGHT * MAX_RAM_REPRESENTATION_WIDTH then
            write_ram(CLEARED_RAM, to_logic_vector(S_RESET_ADDR), S_ACTION_TO_PERFORM_SRAM, S_DATA_IN_SRAM, S_ADDR);
            S_RESET_ADDR <= S_RESET_ADDR + 1; -- set the new address to erase;
        else
            S_RESET <= false; -- tells to stop reset;
            S_MAPPING_IN_RAM <= '1'; -- starts remapping;
        end if;
    elsif S_MAPPING_IN_RAM = '1' then
        if not S_INIT_DONE then -- while the mapping is flowing, proceed writing blocks
            write_block(R_BORDER,
                GAME(S_CURRENT_LEVEL)(S_BLOCKS_ARRAY_COUNTER).x + S_COLUMN_COUNTER,
                GAME(S_CURRENT_LEVEL)(S_BLOCKS_ARRAY_COUNTER).y + S_ROW_COUNTER,
                S_ACTION_TO_PERFORM_SRAM,
                S_DATA_IN_SRAM,
                S_ADDR
            );
        else
            S_MAPPING_IN_RAM <= '0'; -- stop the mapping
            O_IS_UPDATING <= '0'; -- we are not resetting anymore
            O_LEVEL_LOADED_DONE <= '1'; -- confirm new level loading
        end if;
    elsif I_STATE = LOAD_LEVEL then -- if the current state is the changing level state
        S_RESET <= true; -- start soft resetting
        S_RESET_ADDR <= 0; -- reset address reset
        S_CURRENT_LEVEL <= I_CURRENT_LEVEL_NUMBER; -- buffer the current
        S_MAPPING_IN_RAM <= '0'; -- assure the map is not enabled
        O_IS_UPDATING <= '1'; -- communicate we are updating
        elsif I_CE = '1' then -- chip is enabled
            S_CE <= '1';
            S_ADDR <= position_to_address(I_X_TO_CHECK, I_Y_TO_CHECK); -- reads at (x, y) position;
            S_ACTION_TO_PERFORM_SRAM <= '0'; -- reads;
            O_DATA_FROM_MEMORY <= S_DATA_OUT_SRAM; -- outputs the read value;
        elsif I_CE = '0' then
            S_CE <= '0';
        end if;
    end if;
end process;
```



sram_controller

Questo modulo si occupa della gestione della memoria SRAM ed è organizzato nel seguente modo:

```

entity sram_controller is
  port (
    CLOCK      : in std_logic; -- clock in
    RESET      : in std_logic; -- reset async

    DATA_IN    : in std_logic_vector(15 downto 0); -- data in
    DATA_OUT   : out std_logic_vector(15 downto 0); -- data out
    ADDR       : in std_logic_vector(17 downto 0); -- address in

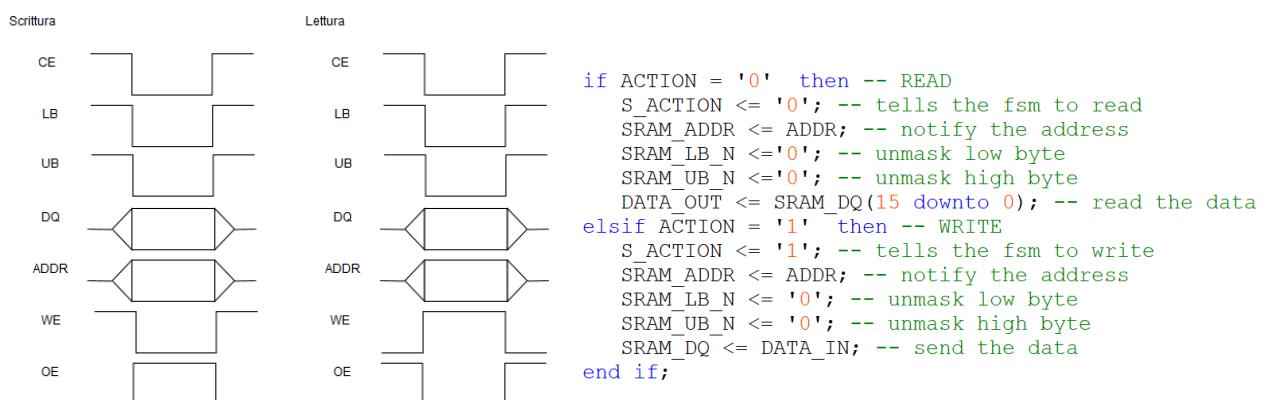
    ACTION     : in std_logic; -- operation to perform
    I_CE       : in std_logic; -- chip select

    SRAM_ADDR  : out std_logic_vector(17 downto 0); -- address out
    SRAM_DQ    : inout std_logic_vector(15 downto 0); -- data in/out
    SRAM_CE_N  : out std_logic; -- chip select
    SRAM_OE_N  : out std_logic; -- output enable
    SRAM_WE_N  : out std_logic; -- write enable
    SRAM_UB_N  : out std_logic; -- upper byte mask
    SRAM_LB_N  : out std_logic -- lower byte mask
  );
end entity;

```

Il modulo implementa semplicemente il protocollo di scrittura e lettura della SRAM. Si occupa quindi di inviare i segnali di controllo *Write Enable*, *Output Enable* e *Chip Enable* per attivare il modulo SRAM.

Seguono le forme d'onda degli input della SRAM nel caso di scrittura e lettura e la rispettiva implementazione.



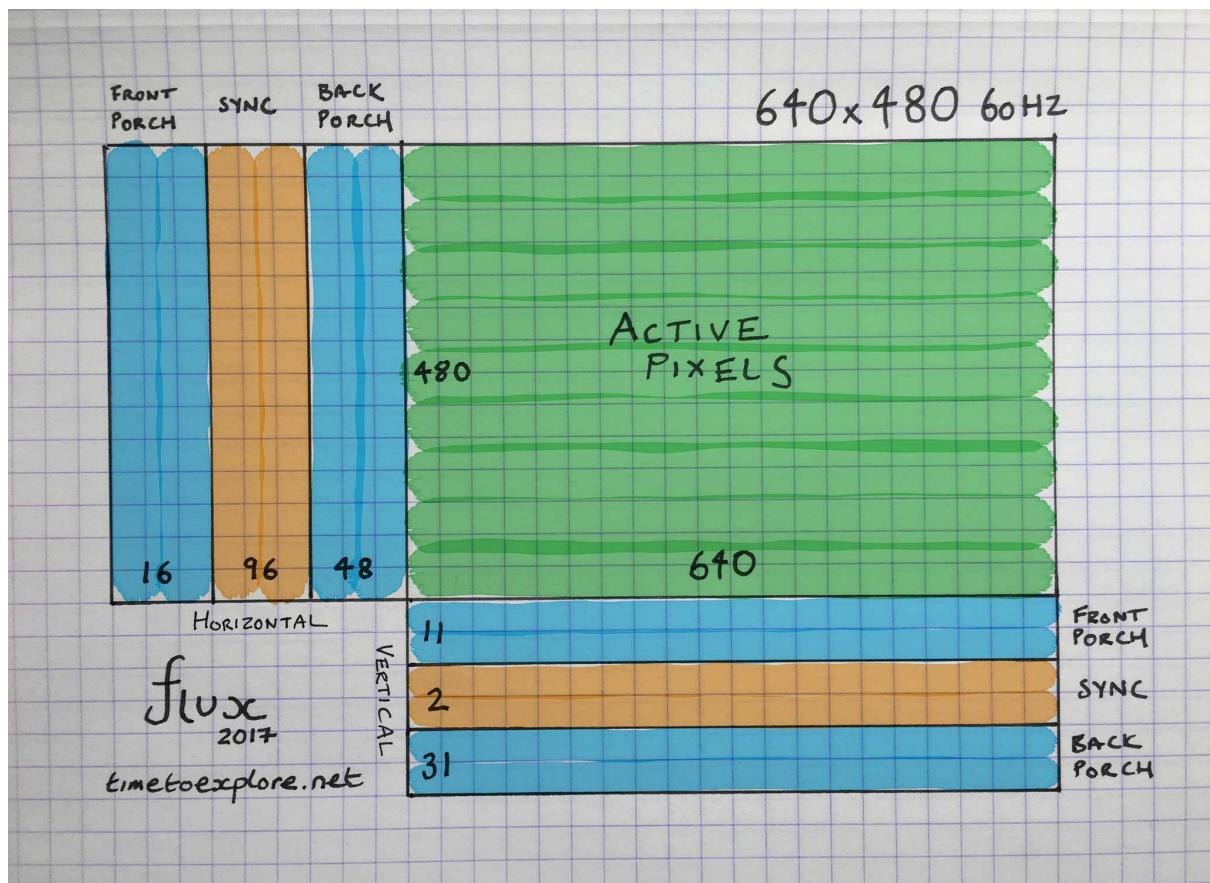
View

Il progetto utilizza la porta VGA presente sulla board per la rappresentazione a video del gioco.

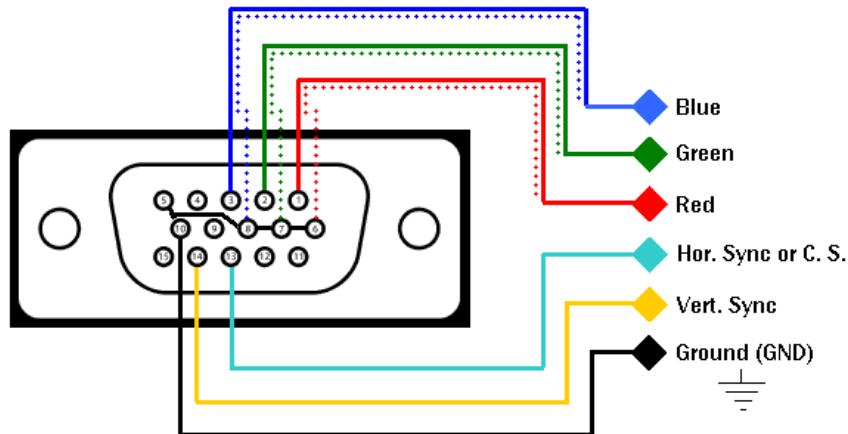
La risoluzione utilizzata è quella standard 640 x 480 pixel visibili con una frequenza di aggiornamento di 60Hz.

Di conseguenza è necessario avere un *Pixel Clock* con una frequenza pari a 25MHz che consente di scansionare per righe (tecnica *Raster Scan*) l'intero schermo 60 volte al secondo considerando anche i "punti" di PORCH e di SYNC non visibili a video.

A partire dal *Pixel Clock* sono anche generati i segnali di sincronizzazione orizzontale e verticale inviati al dispositivo di output.



Per quanto riguarda la porta VGA si hanno 2 pin per la sincronizzazione orizzontale e verticale e 3 pin per le componenti RGB.



I segnali *Horizontal Sync* (HS) e *Vertical Sync* (VS) sono due segnali digitali mentre le tre componenti RGB sono analogiche.

La conversione dal dominio digitale del progetto, al dominio analogico è realizzata da un circuito DAC resistivo che, a differenza del CODEC audio descritto sotto, non richiede alcuna configurazione.

I moduli dedicati alla rappresentazione grafica sono i seguenti:

- *mario_view*
Al suo interno vi è l'intera architettura che gestisce la comunicazione con l'output video
- *video_package*
Contiene tutte le costanti relative alla rappresentazione grafica (tra cui PORCH e SYNC) ed una funzione di utility

mario_view

Questo modulo si occupa di gestire la comunicazione con l'output ed è organizzato nel seguente modo:

```
entity mario_view is
  port
  (
    CLOCK          : in std_logic;
    RESET          : in std_logic;

    -- RICEVUTI DAL DATAPATH
    MARIO_X        : in integer;
    MARIO_Y        : in integer;
    COIN_X         : in integer;
    COIN_Y         : in integer;
    LIVES          : in integer;
    NUM_COIN_CATCHED : in integer;

    -- RICEVUTI DALLA CONTROL UNIT
    CURRENT_LEVEL  : in integer;
    STATE           : in mario_state;
    LEV_LOST        : in std_logic;
    TIMER_VALUE     : in integer;

    -- INVIATI ALLA PORTA
    VGA_HS          : out std_logic;
    VGA_VS          : out std_logic;
    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)
  );
end entity mario_view;
```

L'architettura è realizzata attraverso la rete *DrawProcess* che si occupa di pilotare tutte le uscite.

Sia la generazione degli impulsi di sincronizzazione che la scansione per righe sono realizzate attraverso due contatori *horizontal_pointer* e *vertical_pointer* come viene riportato nel codice VHDL seguente:

```
--Vertical sync
if(vertical_pointer >= VERTICAL_FRONT_PORCH
  and vertical_pointer < VERTICAL_FRONT_PORCH + VERTICAL_SYNC_PULSE) then
  VGA_VS <='1';
else
  VGA_VS <='0';
end if;

--Horizontal sync
if(horizontal_pointer >= HORIZONTAL_FRONT_PORCH
  and horizontal_pointer < HORIZONTAL_FRONT_PORCH + HORIZONTAL_SYNC_PULSE) then
  VGA_HS <='1';
else
  VGA_HS <='0';
end if;
```

```

--update coordinates
if(horizontal_pointer = TOTAL_W-1) then
    if(vertical_pointer = TOTAL_H-1) then
        vertical_pointer <= 0;
    else
        vertical_pointer <= vertical_pointer + 1;
    end if;
    horizontal_pointer <= 0;
else
    horizontal_pointer <= horizontal_pointer + 1;
end if;

```

Dato che *horizontal_pointer* e *vertical_pointer* identificano anche dei punti non visibili sullo schermo, sono state ricavate le variabili *x* e *y* che consentono di identificare i soli pixel visibili sullo schermo.

In termini di codice VHDL, esse sono così definite:

```

x := horizontal_pointer - WINDOW_HORIZONTAL_START;
y := vertical_pointer - WINDOW_VERTICAL_START;

```

Se il puntatore indirizza un pixel dentro l'area visibile, le tre componenti RGB devono assumere un valore che dipende sia dalla posizione del pixel puntato che dai valori provenienti dal *Datapath* e dalla *Control Unit*, altrimenti le uscite RGB devono assumere valore logico zero.

In termini di codice VHDL:

```

if(x >=0 and x < VISIBLE_WIDTH and y >= 0 and y < VISIBLE_HEIGHT) then
    --inside the visible window, draw logic...

else
    --outside visible screen
    VGA_R <= X"0";
    VGA_G <= X"0";
    VGA_B <= X"0";
end if;

```

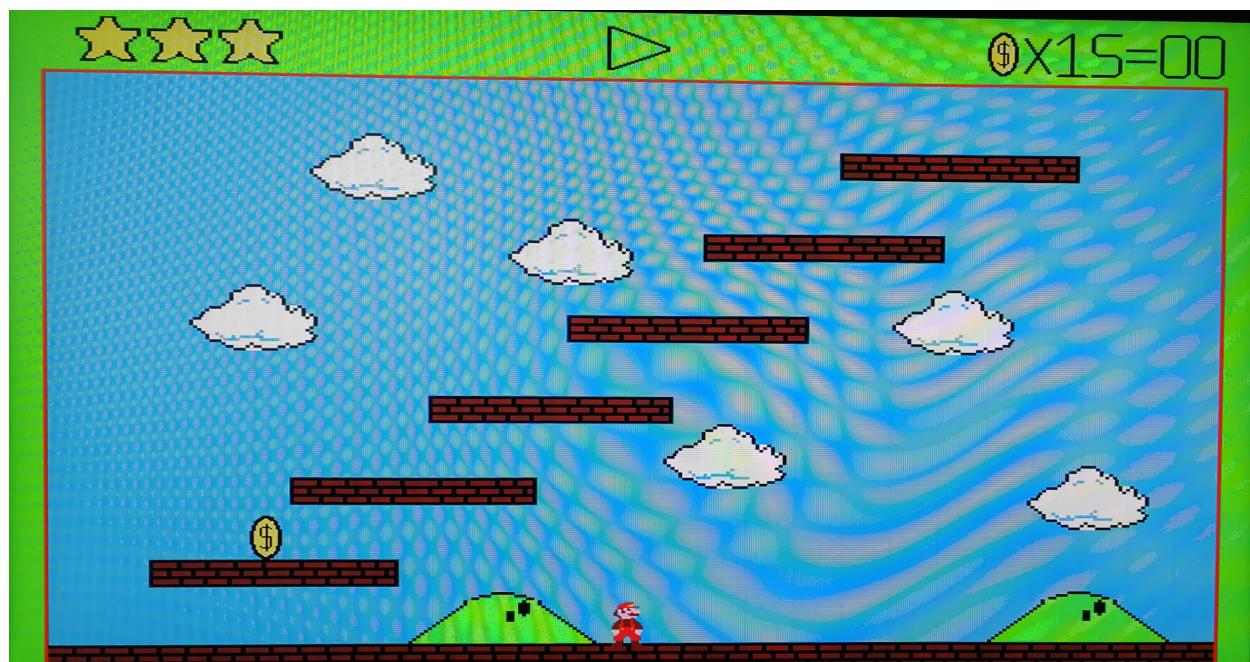
Ad ogni istante di gioco l'immagine mostrata a video è composta dalla barra di stato e dalla finestra di gioco dentro cui Mario si può muovere.

La barra di stato contiene tutte le informazioni che riguardano lo stato della partita corrente, ovvero:

- Vite di Mario
- Countdown durante lo stato PLAY
- Simbolo di pausa durante lo stato PAUSE
- Monete da catturare
- Monete catturate

Un esempio di finestra di gioco è il seguente, e contiene tutti gli elementi grafici utilizzati nel progetto:

- Monete
- Blocchi
- Pavimento
- Scenario di sfondo
 - Colline
 - Nuvole
 - Cielo



Il concetto di profondità è realizzato secondo il comportamento del linguaggio tale per cui scritture multiple, nello stesso periodo di clock, sulla stessa uscita (VGA_R, VGA_G o VGA_B) comportano che questa assuma l'ultimo valore a lei assegnato.

La visualizzazione degli elementi visivi (sprites) è stata realizzata mediante una matrice di valori binari che contiene la definizione dell'elemento che si vuole disegnare. Nello specifico ciascuna matrice definisce un 1 per ogni pixel interno all'elemento e uno 0 per ogni pixel esterno. Per chiarezza è riportata la matrice binaria che descrive i bordi di Mario:

```
type mario_form is array (0 to MARIO_HEIGHT - 1 , 0 to MARIO_WIDTH - 1) of std_logic;
constant mario_borders : mario_form := (
    ("00000001111100000"),
    ("00000111111100000"),
    ("00011111111100000"),
    ("00011111111111000"),
    ("00011111111111100"),
    ("00111111111111100"),
    ("00111111111111110"),
    ("0111111111111110"),
    ("01111111111111100"),
    ("01111111111111110"),
    ("00011111111111000"),
    ("00001111111100000"),
    ("000011111111110000"),
    ("000111111111110000"),
    ("000111111111111000"),
    ("001111111111111000"),
    ("001111111111111100"),
    ("01111111111111110"),
    ("011111111111111100"),
    ("1111111111111111"),
    ("11111111111111111"),
    ("111111111111111111"),
    ("111111111111111111"),
    ("0111111111111110"),
    ("01111111111111110"),
    ("00111111111111100"),
    ("01111110011111110"),
    ("01111100001111110"),
    ("01111100001111110"),
    ("00111100001111100"),
    ("00111100001111100"),
    ("1111110000111111"),
    ("11111100001111111")
```

A ciascuno sprite costituito da diverse componenti colore è stata associata una ulteriore matrice di valori esadecimali che associa ad ogni 1 della matrice binaria la corrispondente componente colore RGB. In questo modo è stato semplicissimo disegnare con un certo livello di dettaglio elementi grafici particolari come lo stesso Mario.

L'estrazione della componente colore contenuta nella matrice è affidata alla funzione *get_color* definita nel package VGA che permette la conversione dalla codifica esadecimale della matrice colore nella corrispondente codifica RGB a 12 bit così definita:

```

function get_color(color : in std_logic_vector(3 downto 0) ) return matrix_color is
variable RGB : matrix_color;
begin
    case color is
        when x"0" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"1" => RGB(0) := "1111"; RGB(1):="1100"; RGB(2):="1101"; --PINK
        when x"2" => RGB(0) := "1111"; RGB(1):="0000"; RGB(2):="0000"; --RED
        when x"3" => RGB(0) := "0111"; RGB(1):="0011"; RGB(2):="0001"; --BROWN LIGHT
        when x"4" => RGB(0) := "1110"; RGB(1):="1101"; RGB(2):="0100"; --YELLOW
        when x"5" => RGB(0) := "0000"; RGB(1):="1111"; RGB(2):="0000"; --GREEN
        when x"6" => RGB(0) := "1111"; RGB(1):="1111"; RGB(2):="1111"; --WHITE
        when x"7" => RGB(0) := "0000"; RGB(1):="1011"; RGB(2):="1111"; --CYAN
        --
        when x"8" => RGB(0) := "0110"; RGB(1):="0001"; RGB(2):="0001"; --BROWN STRONG
        when x"9" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"A" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"B" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"C" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"D" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"E" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
        when x"F" => RGB(0) := "0000"; RGB(1):="0000"; RGB(2):="0000"; --BLACK
    end case;
    return RGB;
end function;

```

A titolo di esempio è riportato il codice che disegna Mario attraverso la matrice binaria *mario_borders* e la matrice colore *mario_colors*:

```
--DRAW MARIO
IF(  x >= MARIO_X and x < MARIO_X + MARIO_WIDTH and
    y >= MARIO_Y and y < MARIO_Y + MARIO_HEIGHT and
    mario_borders(y-MARIO_Y, x-MARIO_X) = '1') then

    colors:=get_color(mario_colors(y-MARIO_Y, x-MARIO_X));
    VGA_R <= colors(0);
    VGA_G <= colors(1);
    VGA_B <= colors(2);
end if;
--END DRAW MARIO
```

Mentre per disegnare un oggetto costituito da una sola componente colore, come ad esempio le cifre numeriche, il codice di esempio è il seguente: il disegno di un oggetto di un unico colore, ad esempio una cifra del numero di monetine da prendere, il codice VHDL è il seguente:

```
-- DRAW NUMBER COIN TO CATCH
--PRIMA CIFRA
if(  x >= COIN_TO_CATCH_FIRST_X and x < COIN_TO_CATCH_FIRST_X + CIFRA_WIDTH and
    y >= COIN_TO_CATCH_FIRST_Y and y < COIN_TO_CATCH_FIRST_Y + CIFRA_HEIGHT and
    CIFRE(NUM_COIN_PER_LEVEL/10)(y-COIN_TO_CATCH_FIRST_Y, x-COIN_TO_CATCH_FIRST_X) = '1') then
    VGA_R <= "0000";
    VGA_G <= "0000";
    VGA_B <= "0000";
end if;
```

CIFRE è un array di 10 matrici di valori binari all'interno del quale sono contenute le rispettive forme di ciascuna cifra.

Per disegnare gli elementi grafici simili, come ad esempio i blocchi o le nuvole, si è utilizzato il costrutto for con condizione di terminazione definita dal numero di elementi da disegnare.

Questo evita la duplicazione di codice e lo rende indipendente dal numero di elementi grafici simili da disegnare.

A titolo di esempio è riportato il codice VHDL per disegnare i blocchi:

```
--DRAW BRICKS
for i in 0 to NUM_BLOCKS - 1 loop
    IF(  x >= GAME(CURRENT_LEVEL)(i).x and x < GAME(CURRENT_LEVEL)(i).x + BLOCK_WIDTH and
        y >= GAME(CURRENT_LEVEL)(i).y and y < GAME(CURRENT_LEVEL)(i).y + BLOCK_HEIGHT) then

        colors:=get_color(brick_colors(y-GAME(CURRENT_LEVEL)(i).y, x-GAME(CURRENT_LEVEL)(i).x));
        VGA_R <= colors(0);
        VGA_G <= colors(1);
        VGA_B <= colors(2);
    end if;
end loop;
--END DRAW BRICKS
```

Audio

Con lo scopo di rendere più piacevole l'esperienza di gioco si è deciso di riprodurre la traccia audio originale di Super Mario Bros attraverso il CODEC AUDIO WM8731 presente sulla board.

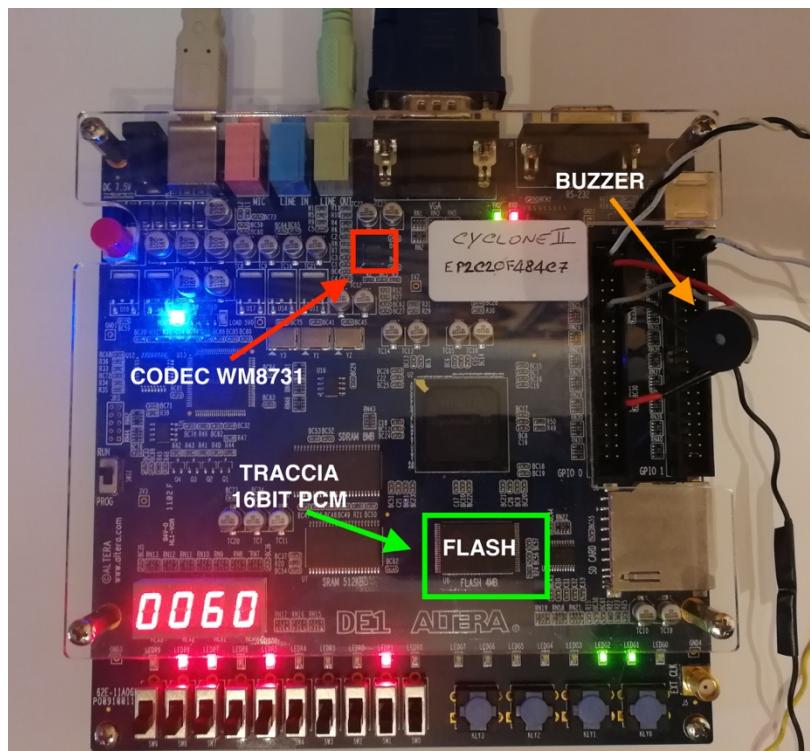
Per la memorizzazione della traccia audio si è deciso di utilizzare la memoria flash di 4MB presente sulla DE1.

Per la riproduzione si è codificata la traccia audio come mono e codifica PCM, con una frequenza di campionamento di 48ksps e una profondità di 16 bit per il campione.

A partire da questi valori numerici si ha che per memorizzare un secondo di traccia audio sono necessari 96KB, di conseguenza in 4 MB di spazio a disposizione è possibile memorizzare solo poco più di 40 secondi.

Per ovviare al limite imposto dalla capacità della flash si è deciso di memorizzare la traccia audio ad una velocità doppia con un conseguente dimezzamento dello spazio di memoria utilizzato.

Oltre al CODEC AUDIO WM8731 è stato utilizzato un Buzzer attivo collegato sul GPIO_0 della DE1 per riprodurre il suono di cattura della moneta.



Codec Audio WM8731

Il codec WM8731 necessita di essere configurato prima del suo utilizzo e per fare ciò si utilizza il protocollo *i2c*

I moduli che contengono la logica per la configurazione e la riproduzione della traccia sono i seguenti:

- *i2c*
Gestisce la configurazione del codec per la riproduzione della traccia
- *audio_mario*
Gestisce l'estrazione della traccia dalla memoria e la conseguente riproduzione tramite il codec

i2c

Il componente *i2c* si occupa dell'inizializzazione dei registri interni del CODEC Audio WM8731

Nell'implementazione del protocollo *i2c*, la FPGA svolge il ruolo di master e si occupa di inviare i dati per la configurazione al CODEC, quest'ultimo svolge il ruolo di slave e risponde con il relativo acknowledge.

Il componente *i2c* è utilizzato solo durante la fase di inizializzazione, mentre a regime non svolge alcuna funzione.

L'entità è organizzata come segue:

```
entity i2c is
port(
    CLOCK_12      : in std_logic;
    RESET         : in std_logic;

    I2C_SCL       : out std_logic;
    I2C_SDA       : inout std_logic;

    DAC_READY     : out std_logic
);
end i2c;
```

All'interno del componente sono definiti i seguenti processi:

- *i2cClockProcess*
questo process si occupa di generare il clock *i2c*
- *FSMI2CProcess*
Si occupa di effettuare una comunicazione con il protocollo *i2c* per inizializzare i singoli registri del codec
- *InitProcess*
Specifica al process *FSMI2CProcess* la configurazione da settare su un determinato registro del codec

i2cClockProcess

In questo processo è definita la logica di generazione di un clock di frequenza compatibile con quella del protocollo *i2c* e regolabile attraverso il parametro I2C_PRESCALER.

Al momento tale frequenza è fissata a 200 kHz.

Il processo genera anche due impulsi, definiti come *clk_en* e *ack_en* che rispettivamente vengono attivati quando *clk_i2c* è al livello logico 0 e quando il *clk_i2c* è al livello logico 1.

Infatti, il protocollo prevede la trasmissione del dato dal master allo slave quando il clock è al valore logico 0 e la trasmissione dell'*ack* quando il clock è al valore logico 1 (si vedano le frecce rosse nelle forme d'onda specificate in seguito).

```
if(clk_prs<I2C_PRESCALER)then
    clk_prs<=clk_prs+1;
else
    clk_prs<=0;
end if;

if(clk_prs<I2C_PRESCALER/2)then ---50 % duty cylce clock for i2c
    clk_i2c<='1';
else
    clk_i2c<='0';
end if;

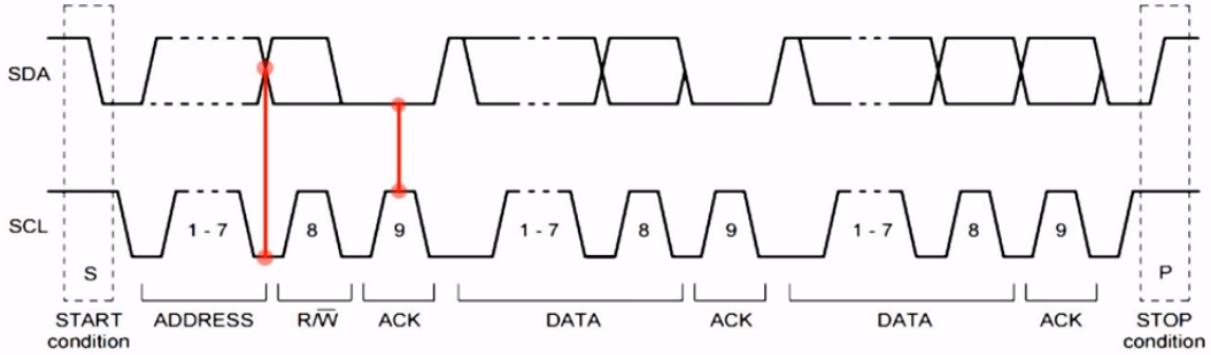
---- clock for ack  on SCL=HIGH
if(clk_prs=I2C_PRESCALER/4)then
    ack_en<='1';
else
    ack_en<='0';
end if;
---- clock for data on SCL=LOW
if(clk_prs=I2C_PRESCALER/2 + I2C_PRESCALER/4)then
    clk_en<='1';
else
    clk_en<='0';
end if;
```

FSMI2CProcess

Questo processo implementa la macchina a stati finiti *i2c*, attraverso la definizione di 9 stati:

```
type fsm is (IDLE, START, ADDR, ACK_ADDR, FIRST, ACK_FIRST, SECOND, ACK_SECOND, STOP);
```

- **IDLE**
Stato di inattività in cui sia il clock che il dato sono al valore logico alto; una volta configurato il CODEC, la macchina *i2c* dovrà trovarsi perennemente in questo stato;
- **START**
Stato in cui il master avvisa lo slave di voler iniziare la comunicazione, portando il dato dal valore logico 1 al valore logico 0;
- **ADDR**
Stato in cui il master scrive sul bus l'indirizzo (7 bit di indirizzo + 1 bit di scrittura) della periferica con cui vuole comunicare, ovvero il CODEC;
- **ACK_ADDR**
Stato in cui si attende l'acknowledge dell'indirizzo da parte dello slave;
- **FIRST**
Stato in cui si inviano i primi 8 bit di dato (7 bit indirizzano il registro del codec su cui scrivere + 1 bit di dato);
- **ACK_FIRST**
Stato in cui si attende l'acknowledge del primo byte;
- **SECOND**
Stato in cui si inviano i secondi 8 bit di dato per la configurazione del registro precedentemente indirizzato;
- **ACK_SECOND**
Stato in cui si attende l'acknowledge del secondo byte;
- **STOP**
Stato in cui il master comunica la fine della transazione *i2c*, portando prima il clock e poi il dato al valore logico 1 e tornando quindi nello stato di IDLE.



Le forme d'onda sopra riportate mostrano la singola transazione I2C per la configurazione di un singolo registro, durante la quale si riceve l'ack positivo coincidente con la lettura da parte del master del valore logico 0 sulla porta dati. Sono inoltre mostrate le condizioni di START e di STOP.

A titolo di esempio è riportato il codice relativo allo stato FIRST che mostra l'invio seriale del dato sulla porta bidirezionale *i2c_sda* che viene messa in alta impedenza quando si deve leggere il bit di acknowledge proveniente dal componente slave.

```

when FIRST=> ---- send 1st 8 bit
  if(data_index>8) then
    data_index<=data_index-1;
    I2C_SDA<=i2c_data(data_index);
  else
    I2C_SDA<=i2c_data(data_index);
    get_ack<='1';
  end if;
  if(get_ack='1')then
    get_ack<='0';
    i2c_fsm<=ACK_FIRST;
    I2C_SDA<='Z';
  end if;

```

InitProcess

Questo processo si occupa di coordinare l'inizializzazione di tutti i registri del CODEC AUDIO.

Ciò è realizzato attraverso il contatore *init_counter* le cui uscite sono collegate ai bit di selezione di un multiplexer. Alle linee di dato del multiplexer sono invece cablate delle stringhe di 16 bit (7 bit di indirizzo del registro + 9 bit di configurazione) contenenti le informazioni specifiche per l'inizializzazione di ogni registro del CODEC (es. CODEC slave, 48ksps, 16 bit di profondità per campione, DAC on, ADC off, ...).

Il processo richiede al processo *FSMI2CProcess* l'esecuzione di una transazione *i2c* tramite il flag *i2c_send_flag*.

Una volta che tutti i registri sono stati inizializzati il componente *i2c* non ha più compiti, la sua uscita *DAC_READY* viene asserita e la riproduzione della traccia audio può iniziare.

Il seguente codice mostra ad esempio, la configurazione per l'attivazione del DAC e la disattivazione dell'ADC.

```
when 2 =>
    --ADC off, DAC on, Linout ON, Power ON
    i2c_data(15 downto 9)<="0000110";
    i2c_data(8 downto 0)<="000000111";
    i2c_send_flag<='1';
```

audio_mario

L'entità *audio_mario* è organizzata come segue:

```
entity audio_mario is
port (
    clockAUDIO : in std_logic;
    RESET      : in std_logic;

    -----WM8731 pins-----
    AUD_BCLK   : out std_logic;
    AUD_XCK    : out std_logic;
    AUD_DACLRCK: out std_logic;
    AUD_DACDAT : out std_logic;

    -----I2C pins-----
    I2C_SCLK   : out std_logic;
    I2C_SDAT   : inout std_logic;

    -----flash pins-----
    FL_ADDR   : out std_logic_vector(21 downto 0);
    FL_DQ     : in std_logic_vector(7 downto 0);
    FL_OE_N   : out std_logic;
    FL_RST_N  : out std_logic;
    FL_WE_N   : out std_logic
);
end entity audio_mario;
```

L'architettura relativa all'entità sopra descritta è implementata istanziando il componente *i2c* e definendo i seguenti process:

- *AudioGenProcess*

Il processo genera degli impulsi ad una frequenza pari alla frequenza di campionamento scelta e invia i campioni da riprodurre;

- *ReadAudioProcess*

Il processo si occupa di estrarre dalla flash i campioni audio memorizzati e li invia al process *AudioGenProcess*

AudioGenProcess

Il processo genera degli impulsi ad una frequenza pari alla frequenza di campionamento scelta in fase di inizializzazione del CODEC (48 kHz).

Tali impulsi sono generati attraverso il counter *audio_prescaler* e sono propagati sulla porta di uscita AUD_DACLRCK.

Ogni volta che viene inviato un impulso, il processo si occupa dell'invio seriale del campione definito da *sample_to_send* attraverso la porta di uscita AUD_DACDAT.

```
if(dac_ready = '1') then
    if(audio_prescaler<AUDIO_PRESCALER_MAX_VALUE)then-----48k sample rate
        audio_prescaler<=audio_prescaler+1;
        clk_en<='0';
    else
        audio_prescaler<=0;
        sample_to_send <= sample_buffer; --get sample
        clk_en<='1';
    end if;

    if(clk_en='1')then-----send new sample
        send_sample_flag<='1';
        data_index<=31;
    end if;

    if(send_sample_flag='1')then
        if(data_index>0)then
            data_index<=data_index-1;
        else
            send_sample_flag<='0';
        end if;
    end if; --if(send_sample_flag='1')
end if; --if(dac_ready = '1')
```

ReadAudioProcess

Il processo si occupa della lettura dalla flash dei campioni memorizzando l'indirizzo corrente nel counter *read_addr*, che, convertito, è propagato alla porta FL_ADDR. Dato che alla FPGA arrivano solamente gli 8 bit meno significativi del bus dati della flash, per leggere un campione sono necessarie due letture.

Per la riproduzione mono della traccia, ogni campione letto è memorizzato sia nella parte superiore che nella parte inferiore del registro *sample_buffer*.

Di seguito il codice che implementa le letture dei campioni dalla memoria:

```
if(dac_ready = '1') then
    if(clk_en='1')then    --48khz

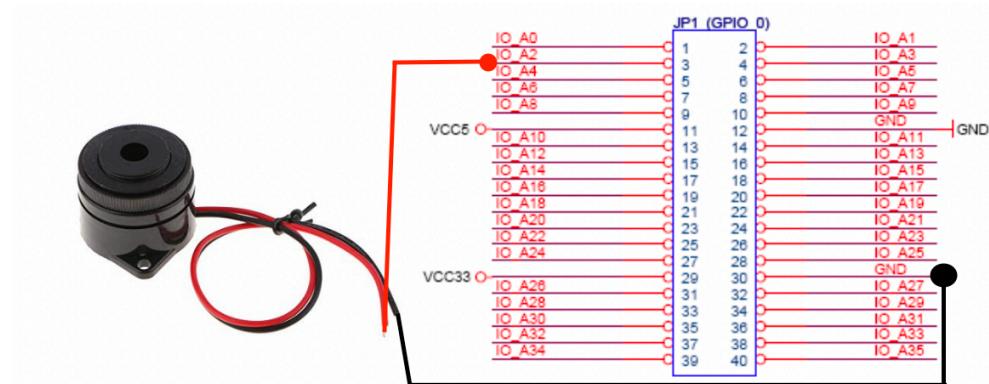
        song_speed_bit_prescaler <= not song_speed_bit_prescaler;

        if(read_addr < LAST_FLASH_ADDR) then
            read_addr <= read_addr + 1;
        else
            read_addr <= 0;
        end if;

        if(song_speed_bit_prescaler = '0')  then
            sample_buffer(7 downto 0) <= FL_DQ;
            sample_buffer(23 downto 16) <= FL_DQ;
        else
            sample_buffer(15 downto 8) <= FL_DQ;
            sample_buffer(31 downto 24) <= FL_DQ;
        end if;
    end if;  --if(clk_en='1')
end if; --if(dac_ready = '1') then
```

Buzzer

Il modulo *Buzzer* implementa la logica per la riproduzione di un segnale acustico ogni volta in cui Mario cattura una moneta.



L'architettura è implementata attraverso il process *CoinCathed* che alla ricezione del segnale COIN_CATCHED proveniente dal *Datapath* asserisce l'uscita BUZZER_PIN per un intervallo di tempo pari a 200ms.

```
if(COIN_CATCHED='1' AND STATE/=PAUSE) then
    start:=true;
end if;

if (start=true) then
    if(one_sec_timer<BEEP_DURATION) then
        one_sec_timer<=one_sec_timer+1;
        BUZZER_PIN<='1';
    else
        one_sec_timer<=0;
        BUZZER_PIN<='0';
        start:=false;
    end if;
end if;
```

Fonti

Altera DE1

https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4904342209-de1-usermanual.pdf

Protocollo SNES

<https://gamefaqs.gamespot.com/snes/916396-super-nintendo/faqs/5395>

Configurazione Codec WM8731

https://www.rockbox.org/wiki/pub/Main/DataSheets/WM8731_8731L.pdf

<https://www.youtube.com/watch?v=zzli7ErWhAA&t=196s>

VGA Timing

<https://timetoeexplore.net/blog/video-timings-vga-720p-1080p>

<http://tinyvga.com/vga-timing/640x480@60Hz>