ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Dipartimento di Informatica – Scienza e Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Progetto finale ed attività progettuale
in
SISTEMI DIGITALI M

Briscola con Altera DE1 Board

CANDIDATI:

Lorenzo Mustich
0000901917

Alessandro Paoletti
0000901759

Alessandro Morabito
0000907048

Anno Accademico: 2019/2020

Sommario

Capitolo 1. La Briscola

"La briscola. Gioco molto semplice. L'avversario sbatte sul tavolo una carta, e voi dovete sbatterla più forte. I buoni giocatori rompono dai quindici ai venti tavoli a partita. […] Quando la carta è abbastanza vecchia, diventa molto dura e pesante, e se non siete allenati è opportuno giocare con guanti da elettricista".

Così nel 1976, lo scrittore bolognese Stefano Benni definiva la briscola nel suo "*Bar Sport*" (Mondadori, 1976). Gioco di carte praticato a tutte le latitudini della Penisola e dalle origini incerte, consiste nel totalizzare un numero di punti più alto rispetto agli altri giocatori.

Si gioca con un mazzo di 40 carte con semi italiani. Alla prima mano, il mazziere di turno distribuisce tre carte ciascuno e lascia una carta scoperta sul tavolo trasversalmente al mazzo, la quale segnerà il seme della briscola e sarà l'ultima ad essere pescata. I punti assegnati ad ogni carta sono così ripartiti:

- Asso -> 11 punti,
- 3 -> 10,

definiti *carichi*;

- Re -> 4,
- Cavallo -> 3,
- Fante (o Donna) -> 2,

dette *figure*;

- 7, 6, 5, 4, 2 –> 0,

detti *lisci*.

Ad inizio partita, il giocatore alla destra del mazziere tira per primo; durante le mani successive, questa prerogativa sarà data al giocatore vincitore della mano precedente.

Il primo giocatore del giro detta, con la propria carta lanciata, il seme della mano: gli altri giocatori potranno "prendere", quindi lanciare per primi alla successiva mano ed, eventualmente, ottenere dei punti se le carte prese non sono dei lisci, se la carta scelta è dello stesso seme della carta dominante, ma con valore maggiore, oppure se la carta giocata è una briscola. Ad ogni turno, i giocatori devono avere sempre tre carte in mano. Il punteggio massimo ottenibile è di 120.

Il numero di giocatori è variabile: dall'uno contro uno alla possibilità di giocare in coppie da due o da tre eliminando via via un numero di carte lisce maggiori (di solito si scartando i due) per evitare che avanzino delle carte.

BriscolaDE1 permette una sfida secca e diretta tra un giocatore reale ed uno virtuale.

Capitolo 2. Algoritmi di gioco

Dall'analisi del problema, è stato possibile tracciare un algoritmo generale della partita:

1. Scegliere chi gioca per primo — JAVA
2. Mischiare le carte del mazzo — JAVA
3. Inviare le tre carte iniziali al giocatore virtuale — JAVA
4. Visualizzare carte giocatore reale, virtuale e briscola — JAVA

for(i = 0; i < 20; i++) {
  if(token == CPU) {
    5. Scegliere la carta da giocare — VHDL
    6. Inviare la carta giocata — VHDL
    7. Attendere il *token* — VHDL
    8. Decidere chi ha preso — VHDL
    9. Assegnare i punti della presa — VHDL
  else {
    5. Attendere il proprio turno (*token*) — VHDL
    6. Scegliere la carta da giocare — VHDL
    7. Inviare la carta scelta — VHDL
    8. Decidere chi ha preso — VHDL
    9. Assegnare i punti della presa — VHDL
  }

10. Confrontare i punti assegnati al giocatore reale e a quello virtuale — VHDL
11. Comunicare l'esito della partita — VHDL

Inoltre, è scaturita la possibilità di suddividere il gioco in due fasi:

- *fase 1*: l'FPGA gioca per prima;
  Se il giocatore virtuale ha delle carte lisce in mano si dovrà comportare come segue:

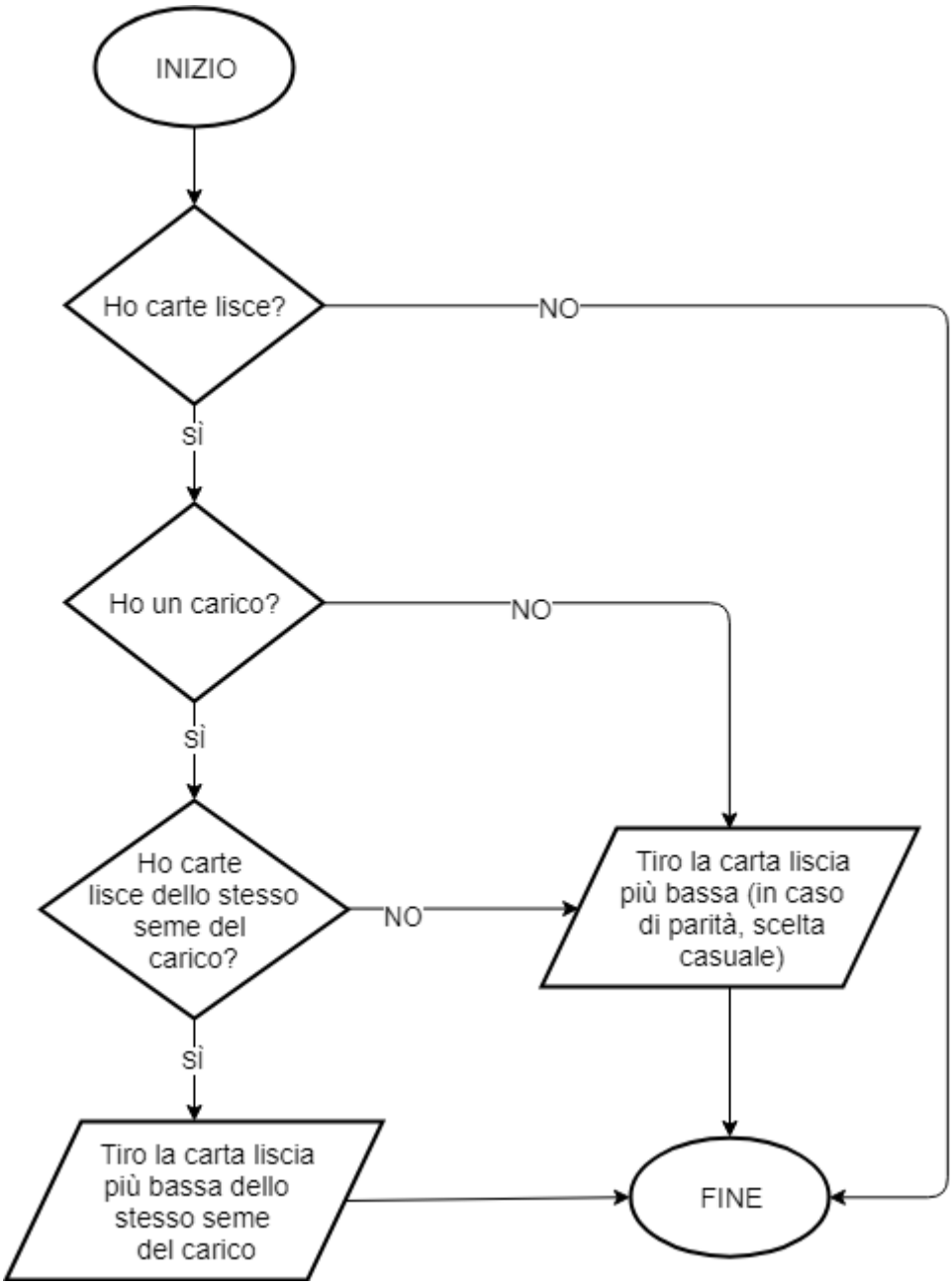

Figura 1. Algoritmo in presenza di carte lisce

In caso di assenza di lisci, sono state definite sette situazioni diverse:

1. *Un solo carico e due figure*;

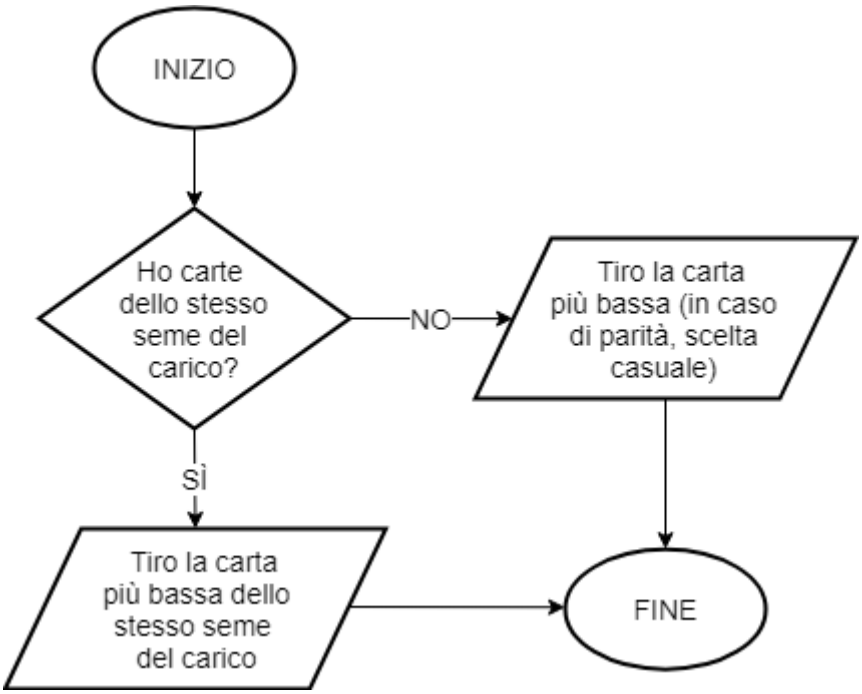

Figura 2. Algoritmo situazione 1

2. *Due carichi e una figura:* scelta forzata, tiro la carta che non è un carico;
3. *Tre carichi*;

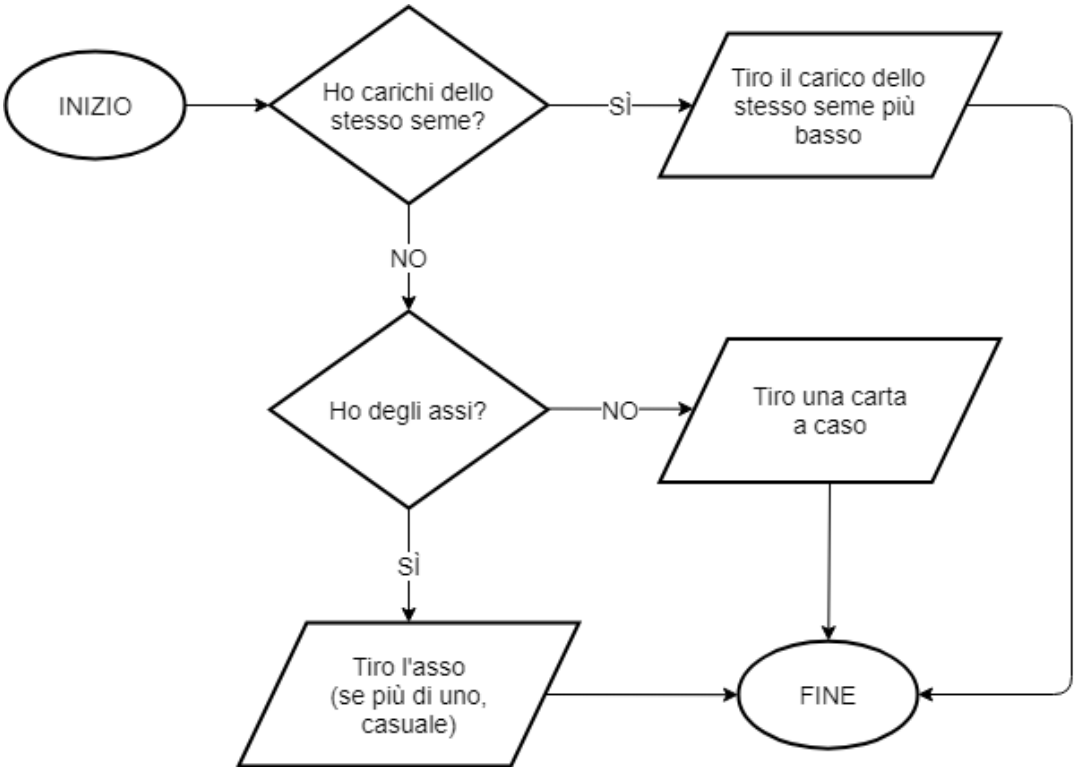

Figura 3. Algoritmo situazione 3

4. *Un carico, una briscola ed una figura*;

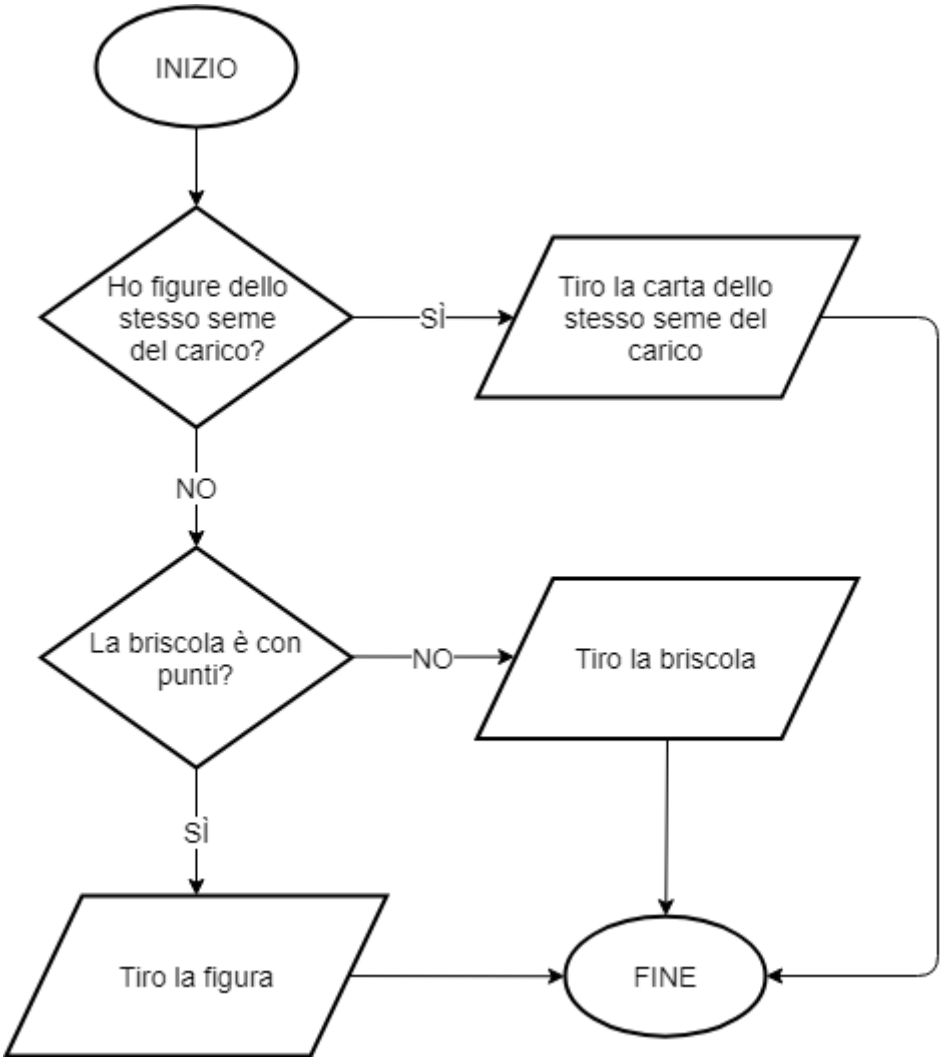

Figura 4. Algoritmo situazione 4

5. *Un carico e due briscole:* scelta forzata, tiro la briscola più bassa;

6. *Due carichi ed una briscola;*

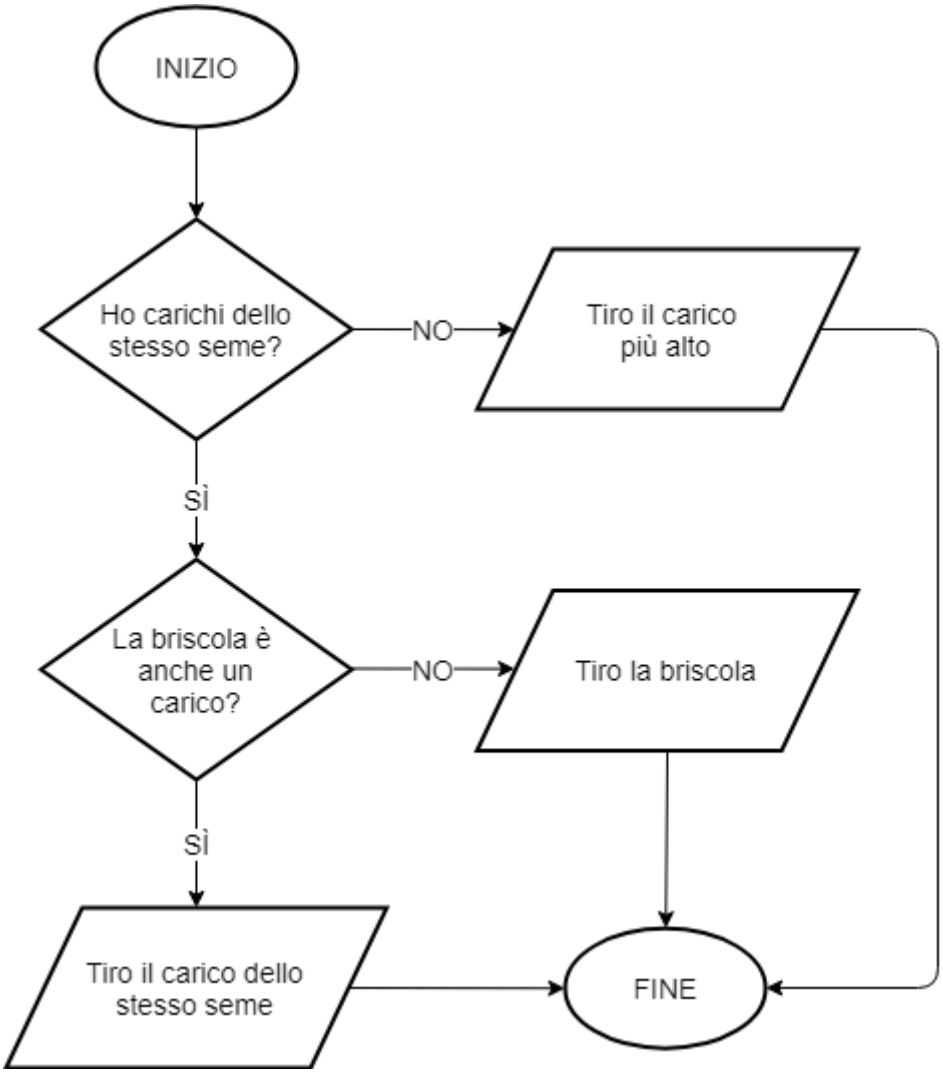

Figura 5. Algoritmo situazione 6

7. *Tre briscole:* tiro la briscola più bassa.

- *fase 2*: l'utente gioca per primo;

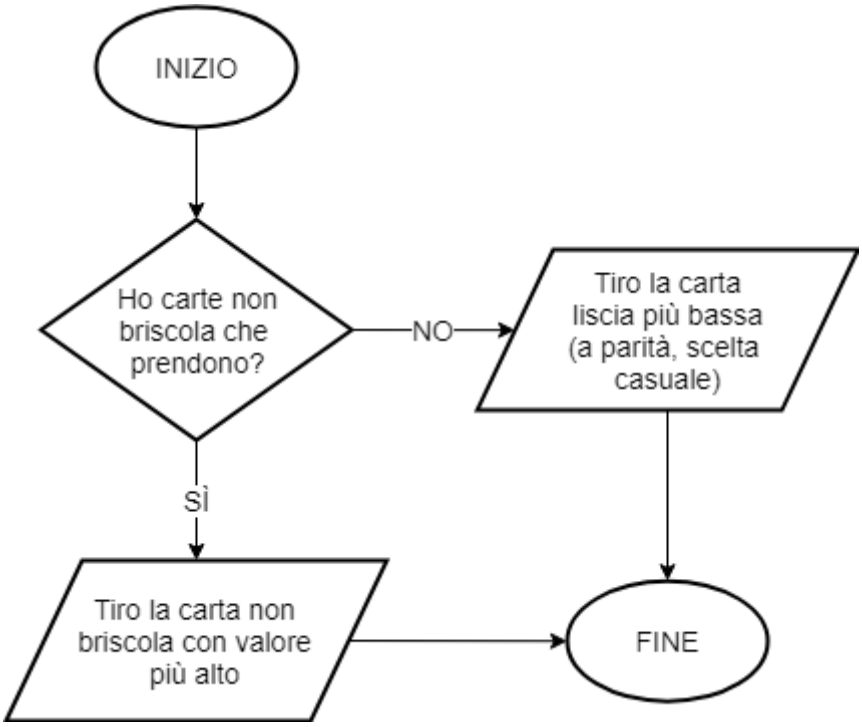

Figura 6. Algoritmo per la scelta della carta in fase 2

Nel caso in cui la mano sia composta da almeno un carico, ci si comporterà come nel caso analogo descritto precedentemente per la fase 1.

Il *penultimo turno* richiede un'analisi approfondita a parte in quanto, a seconda del valore dell'ultima briscola, i giocatori potrebbero attuare strategie diverse rispetto a quelle usate nel corso degli altri turni: se la suddetta briscola è un carico o una figura si tenta di perdere la mano in modo da ottenerla, altrimenti si gioca seguendo le strategie tradizionali.

Capitolo 3. Architettura del sistema

3.1 Entità di top-level

L'*entità di top-level* rappresenta l'entità più astratta di tutto il sistema, la quale si interfaccerà direttamente con la scheda DE1. È stata definita nel file *Briscola.vhd*.

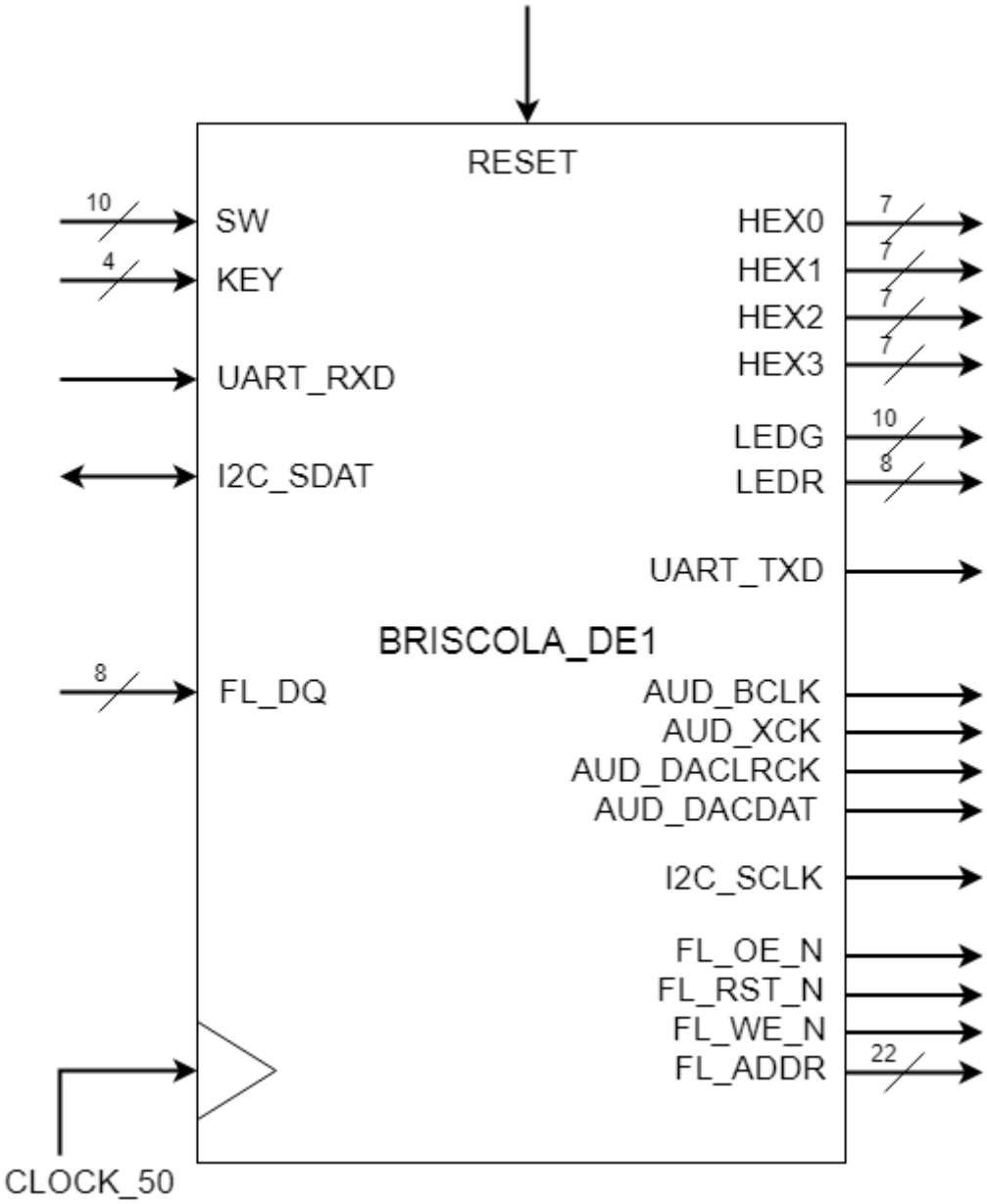

Figura 7. Entità di top-level

Di seguito sono illustrati i pin fisici e il loro utilizzo:

- ***CLOCK_50:*** uno dei tre clock generati dall'FPGA, utilizzato per sincronizzare l'intero sistema;
- ***RESET:*** reset fornito dalla DE1;
- pin per l'interfacciamento con l'utente e per il debug:
  - ***HEX0, HEX1, HEX2, HEX3:*** quattro display a sette segmenti utili in fase di debug per la visualizzazione del valore di segnali, parametri numerici delle carte e identificatore numerico dello stato della Control Unit;
  - ***SW:*** dieci switch, di cui attivi solo ***SW(9)*** per il ***RESET*** e ***SW(7)*** per l'attivazione e lo spegnimento dell'audio;
  - ***LEDG, LEDR:*** dieci led verdi e otto rossi, su cui visualizzare, in fase di debug, i bit ricevuti ed inviati all'applicazione Java;
  - ***KEY:*** quattro tasti utilizzati per abilitare la trasmissione di dati sulla linea seriale;
- pin per la comunicazione seriale:
  - ***UART_RXD*** per la ricezione;
  - ***UART_TXD*** per la trasmissione;
- pin per la riproduzione dell'audio:
  - ***AUD_BCLK:*** clock per la trasmissione dei dati audio;
  - ***AUD_XCK:*** clock per la sincronizzazione del codec audio;
  - ***AUD_DACLRCLK:*** il clock per la sincronizzazione del DAC;
  - ***AUD_DACDAT:*** dati utilizzati dal codec;
- pin per l'utilizzo del protocollo I^2C utile alla configurazione del codec audio:
  - ***I2C_SDAT:*** dati I2C;
  - ***I2C_SCLK:*** clock I2C;
- pin per la lettura da FLASH:
  - ***FL_DQ:*** dati letti;
  - ***FL_OE_N:*** ouput enable (logica negativa);
  - ***FL_WE_N:*** write enable (logica negativa);
  - ***FL_RST_N:*** reset asincrono (logica negativa);
  - ***FL_ADDR:*** indirizzo della cella di memoria letta;

3.2 Architettura

Il *sistema* è basato sulla seguente architettura:

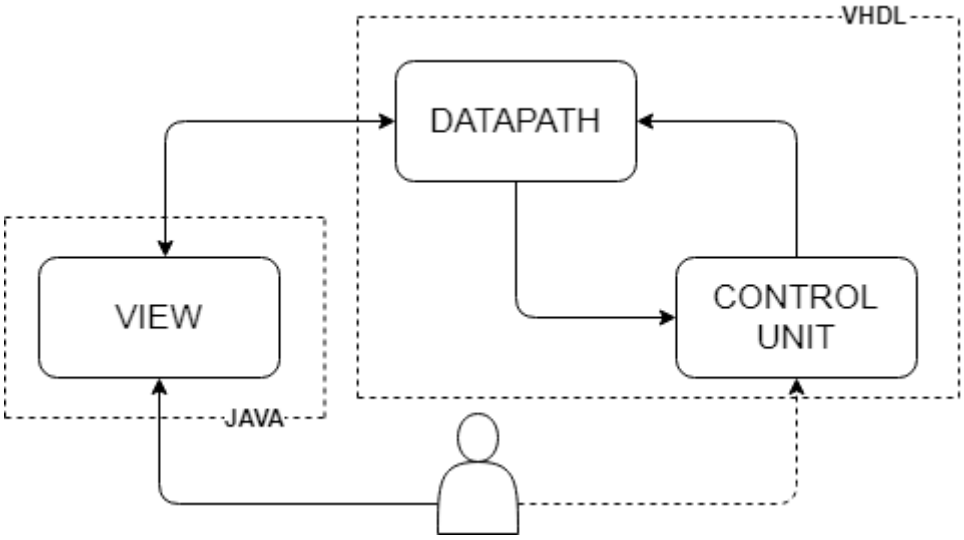

Figura 8. Architettura del sistema

L'*utente* interagisce con un'interfaccia implementata in Java, la quale, attraverso uno scambio di informazioni tramite protocollo seriale, comunica ai moduli VHDL le intenzioni del giocatore. In alcuni punti della partita sarà possibile (a volte necessario) relazionarsi direttamente con l'FPGA.

I moduli principali sono:

- *Briscola_Datapath*: implementa le unità di calcolo e i registri utili al funzionamento del sistema;
- *Briscola_Controller*: si configura come un automa a stati finiti; ha il compito di coordinare tutte le azioni del datapath tramite l'invio e la ricezione di segnali appositi;

ai quali si aggiungono:

- *PLL*: circuito *phase-locked loop* integrato alla scheda; a partire da un clock da 50 MHz, genera due clock da 12 (timing codec audio) e da 50 MHz collegati ad ogni altro modulo del sistema;
- *UART_RX*: implementa l'interfaccia UART necessaria per la comunicazione seriale;
- *Briscola_Audio*: permette l'utilizzo del codec audio WM8731;

3.3 Packages

Nel progetto sono stati inseriti vari package in cui sono stati definiti i tipi di dato, le costanti e le funzioni statiche utilizzate poi in tutti gli altri moduli:

- *briscola_package*: definisce i tipi di dato su cui si basa tutto il progetto;
- *briscola_audio_package*: contiene le costanti utili al funzionamento delle entità relative alla parte audio;
- *briscola_datapath_package*: raccoglie le dichiarazioni e le definizioni delle function legate al datapath;
- *briscola_utility_package*: contiene le dichiarazioni e le funzioni di utilità;

Durante l'analisi del problema, è fuoriuscita la possibilità di suddividere il gioco in varie *fasi* o *"situazioni"*; da qui, la necessità di definire un package per ognuno di essi contenenti dichiarazione ed implementazione di varie funzioni legate al giocatore virtuale:

- *briscola_lisci_package*: da utilizzare se nella mano sono presenti dei lisci;
- *briscola_situazioneNoLisci_package*: da utilizzare se nella mano non vi sono dei lisci;
- *briscola_fase2_package*: da utilizzare quando il giocatore virtuale deve controbattere ad una mossa del giocatore fisico;
- *briscola_penultimo_turno_package*: da utilizzare quando si è arrivati alla penultima mano della partita.

3.3.1 briscola_package

```
-- Costanti --

-- numero di turni totali di una partita (40 carte, 2 giocatori, 20 turni)
constant NUM_TURNI: integer := 20;


-- Tipi di dato --
type vincitore is (GIOCATORE, CPU);
type seme is (BASTONI, DENARI, COPPE, SPADE);


type carta is record
      numero      : integer;
      seme_carta  : seme;
      valore      : integer;
      briscola    : boolean;
end record;
```

```
type mazzo is array (0 to 39) of carta;
type mano_cpu is array (0 to 2) of carta;
```

3.3.2 briscola_audio_package

```
-- Costanti audio --
--
constant LAST_FLASH_ADDR     : positive := 1661635;
constant AUDIO_PRESCALER_MAX : positive := 250;
constant I2C_PRESCALER       : positive := 60;
```

3.3.3 briscola_datapath_package

```
-- Sceglie la carta da giocare (sia in fase 1 che in fase 2)
function decidiCarta(mano : mano_cpu) return integer;

-- Valuta la presa effettuata
function valutaPresa(carta1 : carta; carta2 : carta) return boolean;
```

3.3.4 briscola_utility_package

```
-- Dato un numero ne genera il codice equivalente per un display a 7
-- segmenti
function numberTo7SegmentDisplay(numero : integer) return std_logic_vector;

-- Dato un carattere ne genera il codice equivalente per il display a 7
-- segmenti
function digitTo7SegmentDisplay(carattere : character) return std_logic_vector;

-- Dato il numero della carta ne restituisce il valore
function getValorefromNumber(numero : integer) return integer;

-- Inverte l'ordine dei bit in un qualsiasi vettore
function reverse_vector(a : std_logic_vector) return std_logic_vector;
```

Si è reso necessario definire la funzione *reverse_vector* in quanto il sistema usa dei vettori definiti con codifica *big_endian* mentre il modulo di trasmissione invia stringhe di bit in codifica *little_endian*.

```
-- Controlla se un vettore è pieno di zeri
function vectorIsNotZero(vector : std_logic_vector) return boolean;

-- Data una carta ne restituisce la stringa di byte corrispondente
function fromCartaToByte(cart : carta) return std_logic_vector;
```

3.3.5 briscola_lisci_package

```
-- Determina se nella mano sono presenti dei lisci
function isLiscio(mano : mano_cpu) return boolean;


-- Restituisce true se un carico è presente nella mano, false altrimenti
function isCarico(mano : mano_cpu) return boolean;


-- Restituisce true se esistono un liscio e un carico con lo stesso seme, false
-- altrimenti
function isCartaStessoSemeCarico(mano : mano_cpu) return boolean;


-- Restituisce l'indice della carta liscia più bassa
function getCartaLisciaPiuBassa(mano : mano_cpu) return integer;


-- Restituisce l'indice della carta liscia dello stesso seme nel carico
function getCartaStessoSemeCarico(mano : mano_cpu) return integer;


-- FUNZIONE di TOP-LEVEL --
-- Restituisce l'indice della carta liscia più adeguata
function getCartaLiscia(mano : mano_cpu) return integer;
```

3.3.6 briscola_situazioneNoLisci_package

```
-- SITUAZIONE 1 --

-- Restituisce l'indice della carta più bassa tra quelle presenti nella mano
function getCartaPiuBassa(mano : mano_cpu) return integer;

-- Restituisce l'indice della carta più bassa dello stesso seme di un carico,
-- altrimenti restituisce la più bassa in assoluto
function situazione1(mano : mano_cpu) return integer;

-- SITUAZIONE 2 --

-- Restituisce l'indice della carta che non è un carico (no liscio)
function situazione2(mano : mano_cpu) return integer;

-- SITUAZIONE 3 --
-- Restituisce true se nella mano ci sono più carichi dello stesso seme, false
-- altrimenti
function isCarichiStessoSeme(mano : mano_cpu) return boolean;

-- Restituisce l'indice del carico dello stesso seme più basso
function getCaricoStessoSemePiuBasso(mano : mano_cpu) return integer;
```

```
-- Restituisce true se nella mano sono presenti degli assi, false altrimenti
function getIndiceAssi(mano : mano_cpu) return integer;

-- In caso di più carichi dello stesse seme, restituisce quello più basso
-- altrimenti, se ho assi restituisco l'indice del primo,
-- altrimenti l'indice di una carta casuale
function situazione3(mano : mano_cpu) return integer;

-- SITUAZIONE 4 --

-- Restituisce l'indice della briscola
function getBriscola(mano : mano_cpu) return integer;

-- Restituisce true se la briscola presente è con punti (sia figure che
-- carichi), false altrimenti
function isBriscolaConPunti(mano : mano_cpu) return boolean;

-- Restituisce l'indice della carta non briscola e con pochi punti (una figura)
function getCartaNonBriscolaNonCarico(mano : mano_cpu) return integer;

-- Numero carichi: 1, numero briscole: 1
function situazione4(mano : mano_cpu) return integer;

-- SITUAZIONE 5 --

-- Restituisce l'indice della briscola più bassa
function situazione5(mano : mano_cpu) return integer;

-- SITUAZIONE 6 –

-- Restituisce l'indice del carico più alto
function getCaricoPiuAlto(mano : mano_cpu) return integer;

-- Numero carichi: 2, numero briscole: 1
function situazione6(mano : mano_cpu) return integer;

-- SITUAZIONE 7 --

-- In presenza di tre briscole, restituisce l'indice della più bassa
function situazione7(mano : mano_cpu) return integer;

-- FUNZIONI di TOP_LEVEL --
-- Restituisce il numero di carichi in mano
function getNumeroCarichi(mano : mano_cpu) return integer;

-- Restituisce il numero di briscole in mano
function getNumeroBriscole(mano : mano_cpu) return integer;
```

```
-- Determina le varie situazioni in cui la CPU si può ritrovare in assenza di
-- lisci nella propria mano
function determinaSituazioneNoLisci(num_carichi : integer; num_briscole :
integer; mano : mano_cpu) return integer;
```

3.3.7 briscola_fase2_package

```
-- Determina quale carta sia da lanciare durante la fase 2
function decidiCartaFase2(mano : mano_cpu; cartaTerra : carta) return integer;

-- Restituisce l'indice di una carta non briscola che possa prendere
function getCartaNoBriscolaPresa(mano : mano_cpu; cartaTerra : carta) return
integer;
```

3.3.8 briscola_penultimo_turno_package

```
--
function getBriscolaPiuBassa(mano : mano_cpu) return integer;

--
function getCartaValorePiuAlto(mano : mano_cpu) return integer;

--
function getPrimaCartaLiscia(mano : mano_cpu) return integer;

-- FUNZIONI DI TOP LEVEL –
-- Determina quale carta sia da lanciare
function decidiPenultimo(mano : mano_cpu; briscola_partita : carta) return
integer;

-- Determina quale carta sia da lanciare
function decidiPenultimoFase2(mano : mano_cpu; briscola_partita : carta;
carta_a_terra : carta) return integer;
```

Capitolo 4. Datapath

Il *Datapath* è l'unità centrale di questo progetto sull'FPGA.

Svolge i seguenti compiti:

- riceve, elabora e memorizza i dati ricevuti dall'applicazione Java;
- individua la carta da giocare, secondo le regole espresse nel capitolo 2 e implementate nei package del capitolo 3;
- calcola il risultato delle prese e genera i token corrispondenti;
- trasmette alla GUI i risultati delle prese e delle carte giocate

Di seguito è mostrato lo schema a blocchi dell'entità relativa al *Datapath*:

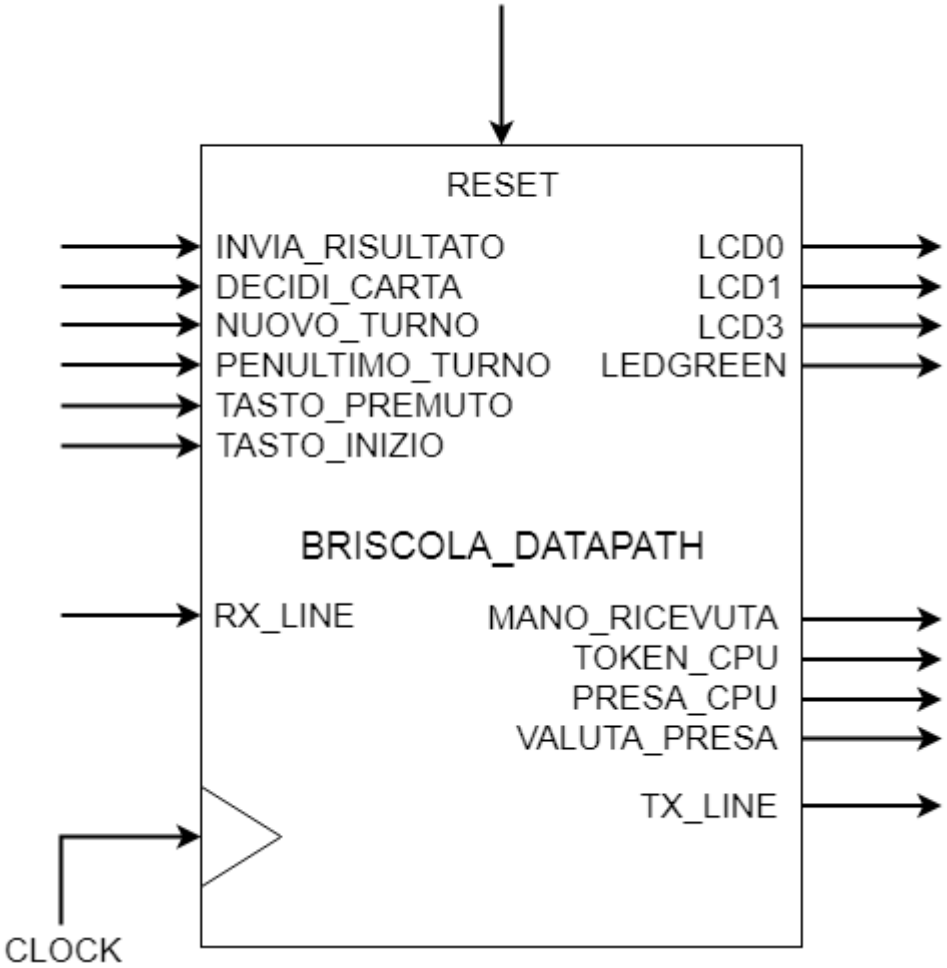

Figura 9. Entità Briscola_Datapath

Il *Datapath* si suddivide in quattro *moduli* che interagiscono tra loro attraverso i segnali generati dalla *Control Unit*: essi svolgono indipendentemente ciascuna delle quattro funzioni descritte in precedenza.

4.1 Tipi di dato

La definizione dei tipi di dato utilizzati è stata fatta all'interno di *briscola_package* (capitolo 3).

4.2 Modulo di ricezione delle carte

Il primo *modulo* abilita la scheda DE1 alla ricezione dei dati dalla porta seriale, attraverso il protocollo RS232, e ne interpreta correttamente il significato, fornendo alla *Control Unit* il segnale di *ricezione della mano*, il quale porterà l'FPGA a scegliere quale carta giocare.

In figura è mostrato lo schema a blocchi del *modulo*:

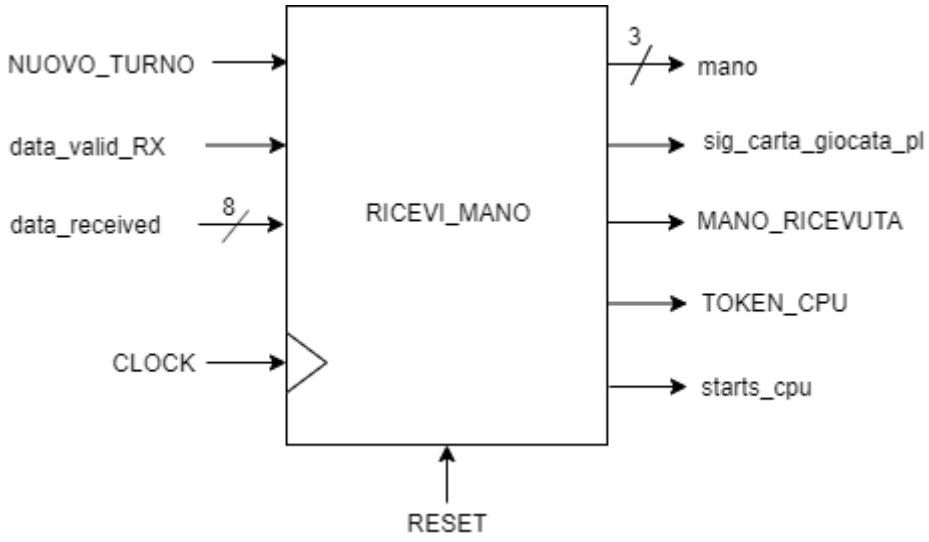

Figura 10. Modulo di ricezione carte

Il componente riceve in ingresso i seguenti segnali:

- ***NUOVO_TURNO:*** generato dalla *Control Unit*, inizializza i segnali interni per l'inizio di un nuovo turno della partita;
- ***data_valid_RX:*** generato dal modulo *UART_RX*, segnala al ricevitore che il pacchetto ricevuto sulla linea seriale è completo e pronto per essere letto;
- ***data_received:*** generato dal modulo *UART_RX*, contiene il dato sotto forma di byte ricevuto dalla porta seriale

e fornisce in uscita i seguenti segnali:

- ***mano:*** array di 3 carte indicante la mano della CPU da cui scegliere la carta da giocare;

- ***sig_carta_giocata_pl:*** contiene la *carta* lanciata dall'utente mediante la GUI; viene utilizzata dagli altri *moduli* per selezionare la carta corretta da lanciare e per la valutare la presa;
- ***MANO_RICEVUTA:*** segnala alla *Control Unit* che la mano iniziale è stata ricevuta;
- ***TOKEN_CPU:*** segnala alla *Control Unit* che è stato ricevuto il *token*;
- ***starts_cpu:*** indica quale giocatore comanda la presa nella mano attuale

Per i segnali riguardanti il *token* ed il suo funzionamento, si rimanda al capitolo 7.

Di seguito sono riportati frammenti di codice significativo per la descrizione del funzionamento del modulo di ricezione delle carte.

```
RiceviMano: process(CLOCK, RESET, NUOVO_TURNO)
      variable carta_reset : carta  := (0, DENARI, 0, false);
begin
      if(rising_edge(CLOCK)) then
            if(NUOVO_TURNO = '1') then
                  token_counter := 0;
                  carta_counter := 0;
            end if;

            if(data_valid_RX = '1') then
                  if(vectorIsNotZero(data_received)) then
                        R_ENABLE <= '1';
                  else
                        R_ENABLE <= '0';
                  end if;
            end if;
      end if;
end process;
```

Il segnale ***R_ENABLE*** indica la presenza di un dato consistente da leggere. Alla sua attivazione si procede a decodificare il dato e trasformarlo in una *carta* o in un *token*.

```
if(R_ENABLE = '1')  then
      if(data_received(7) = '1') then -- è una carta
            if(data_received(0) = '0') then
                  briscola := false;
            else
                  briscola := true;
            end if;
            case data_received(1 to 2) is
                  when "00" =>
                        seme_carta := BASTONI;
                  when "10" =>
                        seme_carta := DENARI;
```

```
                when "01" =>
                      seme_carta := COPPE;
                when "11" =>
                      seme_carta := SPADE;
          end case;

          numero := to_integer(unsigned(reverse_vector(data_received(3 to
    6))));
          valore := getValorefromNumber(numero);
          carta_ricevuta := (numero, seme_carta, valore, briscola);
```

Nella variabile ***carta_ricevuta*** è contenuto il dato di tipo *carta* appena ricevuto dalla porta seriale. In base ai valori assunti dalle variabili interne viene deciso dove memorizzare ogni carta.

```
if((carta_counter > 1) OR (mano_counter > 3 AND stop_ric_mano = '1')) then
      -- carta giocata dal player
      sig_carta_giocata_pl <= carta_ricevuta;
      stop_ric_mano := '0';
      carta_counter := 0;

elsif(carta_in_arrivo = '1') then
      -- carta da mettere nella mano
      mano(indice_carta_giocata) <= carta_ricevuta;
else
      if(stop_ric_mano = '0') then
            if(mano_counter = 3) then
                  -- briscola della partita
                  briscola_partita <= carta_ricevuta;
            elsif(mano_counter < 3) then
                  -- carta nella mano iniziale
                  mano(mano_counter) <= carta_ricevuta;
            end if;
      end if;
end if;

if(stop_ric_mano = '0' AND mano_counter > 3) then
      carta_counter := carta_counter + 1;
end if;

mano_counter := mano_counter + 1;
R_ENABLE <= '0';
```

```
if(mano_counter = 4)  then
      MANO_RICEVUTA <= '1';
      stop_ric_mano := '1';
else
      MANO_RICEVUTA <= '0';
end if;
```

Discorso analogo viene fatto per i *token*, con un'elaborazione più semplice vista la loro configurazione fissa. In caso di ricezione del *reset token* i dati vengono resettati.

```
elsif(data_received(7) = '0') then              -- è un token
      token_counter := token_counter + 1;

      case data_received (0 to 3) is
            when "0101" =>                      -- reset token
                  data_transmitted_token <=
                        ((data_transmitted_token AND "00001111") OR "10100000");
                  AZZERA_TOKEN <= '1';
                  sig_carta_giocata_pl <= carta_reset;
                  token_counter := 0;
                  carta_in_arrivo := '1';
                  carta_counter := 0;

            when others =>
                  AZZERA_TOKEN <= '0';
      end case;
```

In caso di ricezione di un qualsiasi *token*, si considerano i bit relativi al detentore effettivo del *token*, attivando il corrispondente generando un segnale per la *Control Unit*.

```
if(token_counter = 1) then
      case data_received(4 to 6) is
            when "111" =>
                  TOKEN_CPU <= '1';
                  starts_cpu <= true;
            when "000" =>
                  TOKEN_CPU <= '0';
                  starts_cpu <= false;
            when others =>
      end case;
end if;
```

```
case data_received(4 to 6) is
      when "111" =>
            TOKEN_CPU <= '1';
      when "000" =>
            TOKEN_CPU <= '0';
      when others =>
end case;
```

4.3 Modulo di decisione della carta

Il secondo *modulo* ha il compito di scegliere la carta da giocare seguendo le regole di gioco inserite nei package precedentemente descritti. Utilizzando le funzioni di selezione della carta, il modulo inserisce la carta destinata ad essere giocata nel registro apposito, a cui verrà effettuato l'accesso in fase di lettura per l'invio sulla porta seriale alla GUI.

Di seguito lo schema a blocchi del *modulo* in questione:

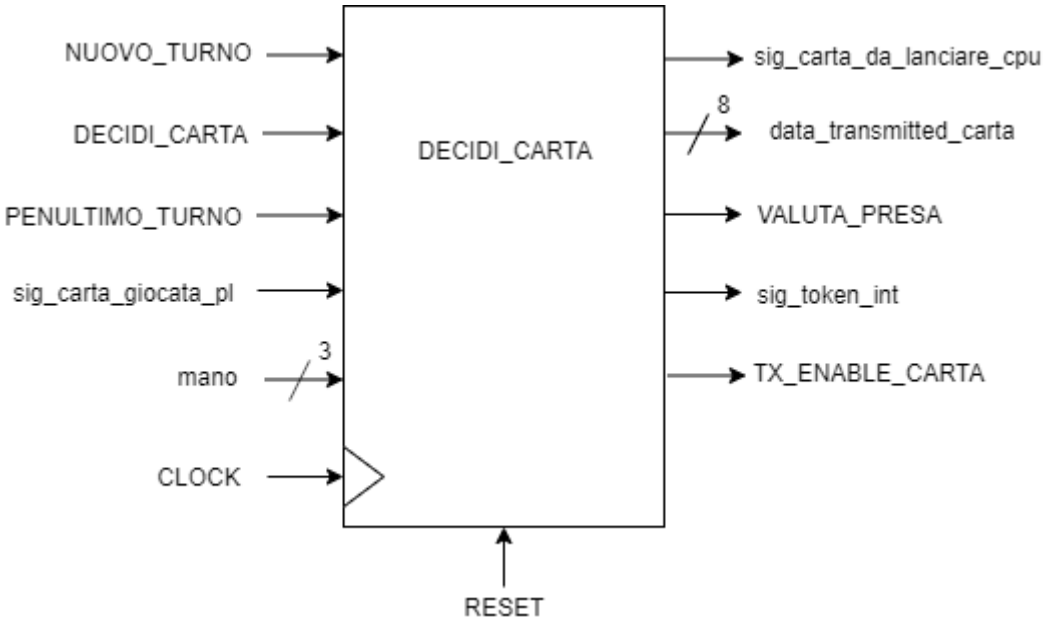

Figura 11. Modulo di decisione della carta

Tralasciando la spiegazione dei segnali già visti nei paragrafi precedenti, il *modulo* prende in ingresso i seguenti segnali:

- ***DECIDI_CARTA:*** generato dalla *Control Unit*, permette al *modulo* di selezionare la carta da lanciare;
- ***PENULTIMO_TURNO:*** generato dalla *Control Unit*, segnala l'inizio del penultimo turno per modificare il comportamento nelle regole di selezione della prossima carta da giocare;

- ***NUOVO_TURNO, sig_carta_giocata_pl, mano:*** già trattati nel *modulo* precedente

e fornisce in output i seguenti segnali:

- ***sig_carta_da_lanciare_cpu:*** registro che contiene la *carta* scelta dalla CPU per essere giocata;
- ***data_transmitted_carta:*** *carta* scelta per essere giocata, trasformata nel formato a 8 bit da inviare alla GUI;
- ***VALUTA_PRESA:*** segnala alla *Control Unit* che la carta è stata scelta e, se l'avversario ha già valutato la sua carta, che è possibile valutare quale giocatore ha vinto il turno;
- ***sig_token_int:*** *token* parziale che viene inizializzato con la configurazione di *token* con presa non valutata se l'avversario non ha giocato la sua carta;
- ***TX_ENABLE_CARTA:*** segnala al *modulo di trasmissione dei dati* sulla porta seriale che il dato presente in ***data_trasmitted_carta*** è pronto per essere inviato alla GUI.

Di seguito vengono riportati frammenti di codice significativo per descrivere il funzionamento del modulo di decisione della carta da giocare.

Con il segnale ***NUOVO_TURNO*** attivo, si resettano la variabile di appoggio ***carta_da_lanciare*** e il corrispondente registro.

```
DecidiCarta: process(CLOCK, DECIDI_CARTA, PENULTIMO_TURNO, NUOVO_TURNO)
begin
	if(rising_edge(CLOCK)) then
		if(NUOVO_TURNO = '1') then
			carta_da_lanciare := carta_reset;
			sig_carta_da_lanciare_cpu <= carta_reset;
		end if;
```

Il segnale ***PENULTIMO_TURNO*** indica che la partita si trova al penultimo turno in cui è necessario un cambio di strategia, con regole diverse da quelle della altri mani.

```
	if(DECIDI_CARTA = '1') then
		if(PENULTIMO_TURNO = '1') then
			indice := decidiPenultimo(mano, briscola_partita);
```

Il registro contenente la carta lanciata dall'utente è utile a discriminare se ci si trova durante la fase 1 o la fase 2:

- se contiene ancora il valore *carta reset*, il giocatore non ha ancora lanciato, perciò si sceglierà la carta seguendo il set di regole della fase 1;
- se contiene un valore valida vorrà dire che il giocatore ha già lanciato una carta e che quindi si seguiranno le regole della fase 2.

La *carta* scelta viene individuata utilizzando il suo indice relativo all'array corrispondente alla mano dell'FPGA.

```
if(sig_carta_giocata_pl.numero = 0) then
	if(isLiscio(mano)) then
		indice := getCartaLiscia(mano);
	else
		num_briscole := getNumeroBriscole(mano);
		num_carichi := getNumeroCarichi(mano);
		indice := determinaSituazioneNoLisci(num_carichi, num_briscole,
			mano);
	end if;
else
	indice := decidiCartaFase2(mano, sig_carta_giocata_pl);
end if;
```

La *carta* viene poi posta nel registro apposito e, se il giocatore umano ha già lanciato la sua carta, viene abilitato il segnale ***VALUTA_PRESA***. La funzione ***fromCartaToByte***, presente nel package ***briscola_utility_package***, consente di tradurre la carta in un byte pronto da inviare.

```
indice_carta_giocata <= indice;
carta_da_lanciare := mano(indice);
sig_carta_da_lanciare_cpu <= carta_da_lanciare;
data_transmitted_carta <= fromCartaToByte(carta_da_lanciare);
TX_ENABLE_CARTA <= TASTO_PREMUTO;

if(sig_carta_giocata_pl.numero > 0) then
	sig_token_int <= "00000000";
	VALUTA_PRESA <= '1';
else
	sig_token_int <= reverse_vector("01110101");
	VALUTA_PRESA <= '0';
end if;
```

4.4 Modulo di valutazione della presa

Il terzo *modulo* è utilizzato per valutare le due carte lanciate dai due giocatori e decidere chi dei due ha vinto il turno corrente, incrementandone i punteggi e predisponendo il *token* contenente l'informazione appena elaborata.

Di seguito lo schema a blocchi del *modulo* in questione:

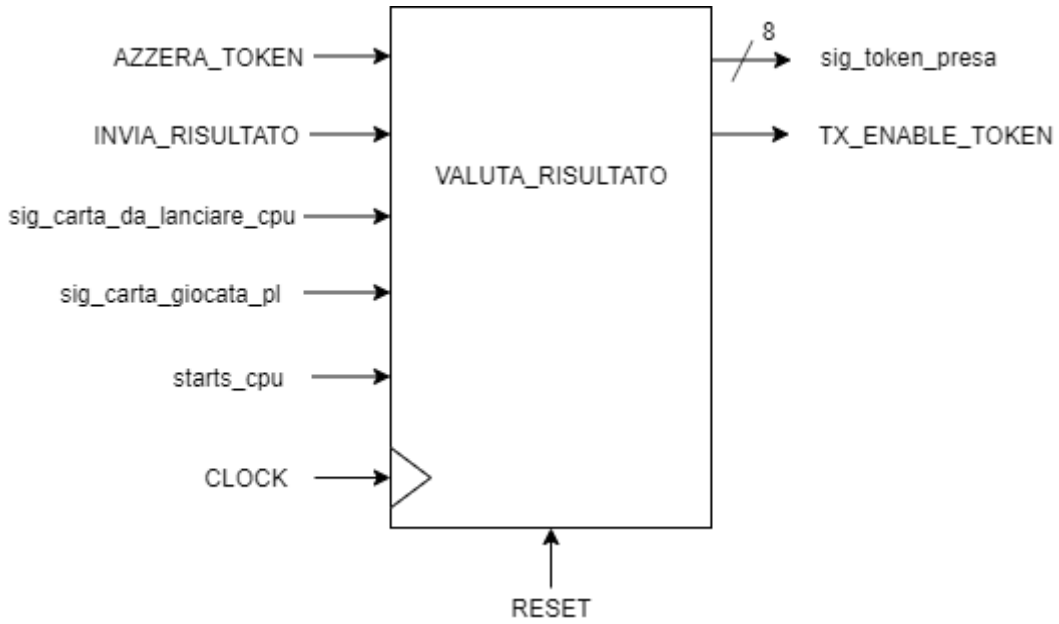

Figura 12. Modulo per la valutazione della presa

Tralasciando la spiegazione dei segnali già visti nei paragrafi precedenti, il *modulo* prende in ingresso i seguenti segnali:

- ***INVIA_RISULTATO:*** generato dalla *Control Unit*, indica l'inizio delle operazioni di valutazione del risultato del turno;
- ***sig_carta_da_lanciare_cpu, sig_carta_giocata_pl:*** registri contenenti le *carte* lanciate dai due giocatori;
- ***starts_cpu:*** indica quale dei due giocatori abbia iniziato il turno per primo, utilizzato in fase di valutazione per determinare chi comanda la presa.

e fornisce in output i seguenti segnali:

- ***sig_token_presa:*** *token* parziale che viene inizializzato con la configurazione corrispondente al risultato della presa
- ***TX_ENABLE_TOKEN:*** segnala al *modulo di trasmissione dei dati* sulla porta seriale che il dato presente in ***data_trasmitted_token*** è pronto per essere trasmesso alla GUI.

Di seguito vengono riportati frammenti di codice significativo per descrivere il funzionamento del modulo di valutazione della presa.

Quando entrambe le carte sono state lanciate, si testa il valore del segnale booleano ***starts_cpu***: data la funzione di valutazione della presa *valutaPresa*

- se indica come vincente la CPU (*risultato = true*), si sommano i punti delle carte giocate da entrambi i giocatori nella relativa variabile accumulatrice e si predispone il *token* con la configurazione corrispondente;
- se è l'utente ad aver vinto la mano (*risultato = false*), i punti delle carte giocate da entrambi i giocatori saranno aggiunti alla variabile corrispondente

```
ValutaRisultato : process(CLOCK, INVIA_RISULTATO) is
begin
      if(rising_edge(CLOCK)) then
            if(INVIA_RISULTATO = '1') then
                  if((sig_carta_da_lanciare_cpu.numero > 0) AND
                        (sig_carta_giocata_pl.numero > 0)) then
                        if(starts_cpu) then
                              risultato :=
                                    valutaPresa(sig_carta_da_lanciare_cpu,
                                                      sig_carta_giocata_pl);
                              if(risultato) then
                                    punti_cpu <= punti_cpu +
                                          sig_carta_da_lanciare_cpu.valore +
                                          sig_carta_giocata_pl.valore;
                                    -- non tocca al giocatore, ha preso la CPU
                                    byte_result := "00001111";
                              else
                                    punti_player <= punti_player +
                                          sig_carta_da_lanciare_cpu.valore +
                                          sig_carta_giocata_pl.valore;
                                    -- tocca al giocatore, ha preso il
                                    -- giocatore
                                    byte_result := "01110000";
                              end if;
```

Qualora il valore del segnale ***starts_cpu*** sia falso (all'inizio del turno il giocatore umano è di mano), le operazioni sono speculari a quelle viste precedentemente.

```
        else
              risultato := valutaPresa(sig_carta_giocata_pl,
                          sig_carta_da_lanciare_cpu);
              if(risultato) then
                    punti_player <= punti_player +
                           sig_carta_da_lanciare_cpu.valore +
                           sig_carta_giocata_pl.valore;
                    -- tocca al giocatore, ha preso il giocatore
                    byte_result := "01110000";
              else
                    punti_cpu <= punti_cpu +
                          sig_carta_da_lanciare_cpu.valore +
                          sig_carta_giocata_pl.valore;
                    -- non tocca al giocatore, ha preso la CPU
                    byte_result := "00001111";
              end if;
        end if;
```

Al termine della valutazione della presa, viene inizializzato il segnale di *enable* per la trasmissione del *token* e il *token* finale viene posto nell'apposito registro (***sig_token_presa***).

Se uno dei giocatori non ha ancora lanciato la sua carta, la variabile di appoggio ***byte_result*** viene azzerata.

```
              TX_ENABLE_TOKEN <= INVIA_RISULTATO;
              sig_token_presa <= reverse_vector(byte_result);
      else
              byte_result := "00000000";
      end if;
```

4.5 Modulo di trasmissione dei dati

Il quarto e ultimo *modulo* si occupa di inviare i dati sulla porta seriale.

A differenza di quanto fatto per la ricezione per cui è stato definito un *modulo esterno* che fornisca il *flag* di *dato completo* e il *byte ricevuto*, in questo *modulo* la trasmissione dei bit viene effettuata direttamente dal *modulo* stesso che si occupa della composizione del *pacchetto* e della sincronizzazione con il *baud rate*.

Di seguito lo schema a blocchi del *modulo* in questione:

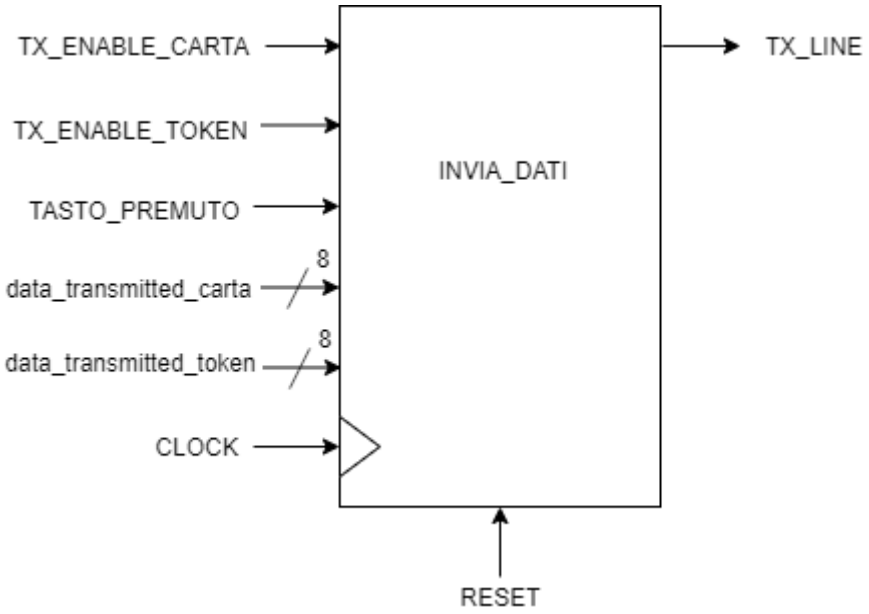

Figura 13.

Tralasciando la spiegazione dei segnali già visti nei paragrafi precedenti, il *modulo* prende in ingresso i seguenti segnali:

- ***TX_ENABLE_CARTA, TX_ENABLE_TOKEN:*** segnali di *enable* che indicano la validità dei byte corrispondenti alla *carta* e al *token* da inviare;
- ***TASTO_PREMUTO:*** proveniente dal pin ***KEY(1)*** della board, indica l'evento di pressione del tasto;
- ***data_transmitted_carta, data_transmitted_token:*** byte corrispondenti alla *carta* e al *token* da inviare sulla linea seriale

e fornisce in output i seguenti segnali:

- ***TX_LINE:*** pin collegato alla porta seriale di trasmissione (***UART_TXD***)

Di seguito vengono riportati frammenti di codice significativo per descrivere il funzionamento del *modulo di trasmissione dei dati*.

Per la trasmissione dei byte sono necessarie due variabili così definiti:

- contatore (***count*** e ***count_tk***): range da 0 a 5207, calcolato come il rapporto tra il *clock interno* utilizzato e il *baud rate* scelto ($50\ MHz\ /\ 9600\ bit/s$); il suo valore massimo indica quanti colpi di clock sono necessari per inviare un bit;
- indice (***bit_number*** e ***bit_number_tk***): range da 0 a 10, viene incrementato ogni volta che il contatore raggiunge il suo valore massimo e indica la posizione del bit da inviare all'interno del registro che lo contiene

```
InviaByte : process(CLOCK, TX_ENABLE_CARTA, TX_ENABLE_TOKEN) is
      -- Variabili per la trasmissione --
      -- 9600 baud generator variable (50MHz/9600)
      variable count : integer range 0 to 5207 := 5207;
      -- start bit + 8 data bits + stop bit => 10 bits
      variable bit_number : integer range 0 to 10 := 0;
      -- 9600 baud generator variable (50MHz/9600) (for token)
      variable count_tk : integer range 0 to 5207 := 5207;
      -- start bit + 8 data bits + stop bit => 10 bits (for token)
      variable bit_number_tk  : integer range 0 to 10 := 0;
```

La trasmissione viene attivata quando i segnali ***TX_ENABLE_CARTA*** e ***TASTO_PREMUTO*** sono entrambi attivi. Ad ogni reset del contatore viene incrementato il ***bit_number***: quando esso è settato a 0 viene inviato lo start bit; quando è al valore 9 viene inviato lo stop bit; quando è all'interno del range 1-8 vengono inviati i bit dei dati effettivi.

Il *flag* ***byte_not_sent*** permette di inviare una carta alla volta ad ogni pressione del tasto: senza di esso la trasmissione dei dati sarebbe continua, inviando 9600 bit/s per ogni secondo quando il tasto è premuto.

```
      if(TX_ENABLE_CARTA = '1' AND TASTO_PREMUTO = '1') then
            if(count = 5207 AND byte_not_sent) then
                  if(bit_number = 0) then
                        TX_LINE <= '0';         --start bit
                  elsif(bit_number = 9) then
                        TX_LINE <= '1';         -- stop bit
                  elsif((bit_number > 0) AND (bit_number < 9)) then
                        -- 8 data bits
                        TX_LINE <= data_transmitted_carta(bit_number-1);
                  end if;

                  bit_number := bit_number + 1;
```

```
                if(bit_number = 10) then
                        --resetting the bit number
                        byte_not_sent := false;
                        bit_number := 0;
                        card_sent := true;
                end if;
        end if;

        count := count + 1;

        if(count = 5208) then
                --resetting the baud generator counter
                count := 0;
        end if;
    else
        byte_not_sent := true;
    end if;
```

Lo stesso codice di invio dei dati viene usato per il *token*, imponendo il vincolo di precedenza nell'invio della carta con la variabile ***card_sent***.

```
if(TX_ENABLE_TOKEN = '1' AND TASTO_PREMUTO = '1') then
    if(count_tk = 5207 AND byte_not_sent_tk) then
        if(card_sent) then
            if(bit_number_tk = 0) then
                TX_LINE <= '0';         --start bit
            elsif(bit_number_tk = 9) then
                TX_LINE <= '1';         -- stop bit
            elsif((bit_number_tk > 0) AND (bit_number_tk < 9)) then
                --8 data bits
                TX_LINE <= data_transmitted_token(bit_number_tk-1);
            end if;

            bit_number_tk := bit_number_tk + 1;

            if(bit_number_tk = 10) then   --resetting the bit number
                byte_not_sent_tk := false;
                bit_number_tk := 0;
                card_sent := false;
            end if;
        end if;
    end if;

    count_tk := count_tk + 1;

    if(count_tk = 5208) then  --resetting the baud generator counter
        count_tk := 0;
    end if;
```

Capitolo 5. Control Unit

La *Control Unit* contiene lo stato attuale del *gioco* e si occupa dell'avanzamento di stato del *sistema*. Tramite opportuni segnali di controllo, regola le attività svolte dal *Datapath*.

Lo schema a blocchi dell'entità *Briscola_Controller* è il seguente:

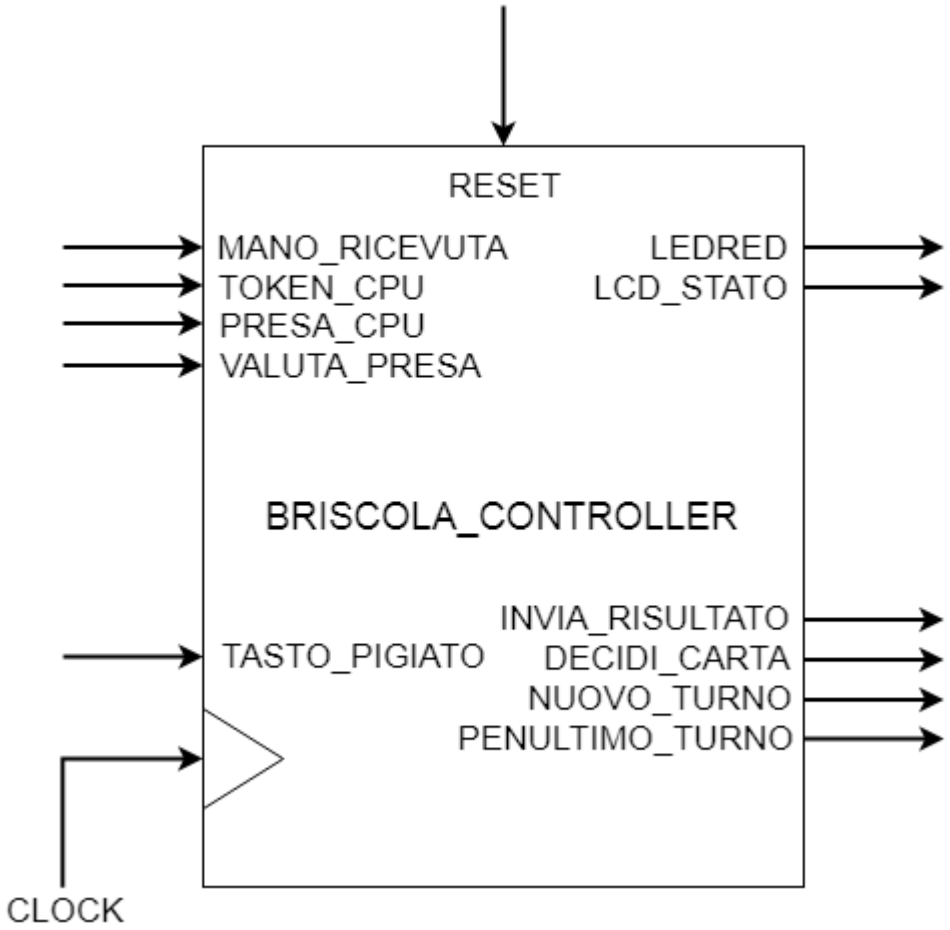

Figura 14. Entità Briscola_Controller

Il *modulo* trattato in questo capitolo è stato progettato come un *automa a stati finiti*:

```
type s_briscola is (S_IDLE, S_MANO_RICEVUTA, S_DECIDI_LANCIA_CARTA,
                    S_ATTENDI_TOKEN, S_INVIA_RISULTATO, S_DECRETA_VINCITORE);
```

Di seguito è mostrato il *diagramma degli stati* del *sistema*:

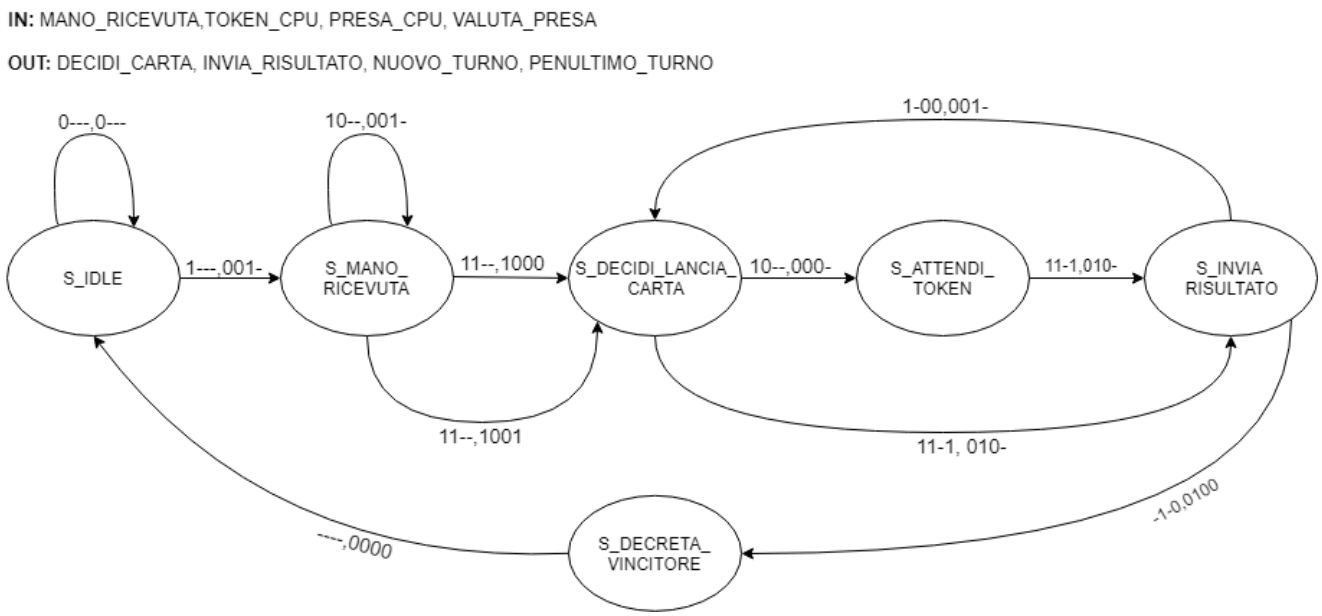

Figura 15. Diagramma degli stati

- *S_IDLE:* rappresenta lo stato iniziale, innescato all'accensione, in cui i segnali ***DECIDI_CARTA***, ***INVIA_RISULTATO***, ***NUOVO_TURNO***, ***PENULTIMO_TURNO***, generati per il *Datapath,* vengono resettati.
  Si tratta di uno stato di transizione che commuta nello stato *MANO_RICEVUTA* nel momento in cui sono state ricevute le carte da memorizzare nella mano e la briscola della partita.
  Durante lo stato di *IDLE*, il *Datapath* è in ascolto sulla linea seriale in attesa di ricevere le tre carte della mano; avvenuta la ricezione, inizia un nuovo turno della partita, si attiva il segnale ***NUOVO_TURNO*** per il *Datapath* e viene incrementato il contatore interno per il numero di turni della partita;
- *S_MANO_RICEVUTA:* questo stato rappresenta uno stato di stallo in cui si segnala l'avvenuta ricezione della carta e si aspetta di ricevere il *token* dalla GUI. Il *token* può essere ricevuto o subito dopo l'invio della mano qualora il giocatore abbia scelto di far iniziare la CPU, o contemporaneamente all'invio della carta lanciata dal giocatore.
  Il *Datapath* segnala la ricezione del *token* ponendo il segnale ***TOKEN_CPU*** a livello alto, il quale fa commutare la *Control Unit* nello *stato di decisione della carta*, inviando al *Datapath* il segnale di ***DECIDI_CARTA***.
  In questo stato viene utilizzata una variabile interna, ***num_turni***, necessaria per discriminare il comportamento tra la modalità di gioco classica e quella ad hoc per il penultimo turno: quando la variabile ***num_turni*** arriva al valore 17,

viene asserito il segnale ***PENULTIMO_TURNO*** che indica al *Datapath* la modifica del comportamento in questo speciale turno;

- *S_DECIDI_E_LANCIA_CARTA:* in questo stato, mentre il *Datapath* ha già scelto la carta da giocare, il *Controller* aspetta di mutare lo stato in base all'avanzamento della partita: quando è l'FPGA a lanciare la carta per prima, alla pressione del tasto di invio carta essa perde il *token* e commuta nello *stato di attesa del token*; viceversa, se è stato il giocatore a lanciare per primo la carta nel turno attuale, il *Datapath* asserisce il segnale ***VALUTA_PRESA*** alla ricezione della carta avversaria e commuta nello *stato di invio del risultato* contestualmente alla ricezione del *token*, inviando il segnale ***INVIA_RISULTATO*** al *Datapath* per il calcolo della presa;
- *S_ATTENDO_TOKEN:* in questo stato, simile allo stato *IDLE*, la *Control Unit* è in attesa della ricezione del segnale ***TOKEN_CPU*** generato dal *Datapath*, cui corrisponde la ricezione del *token* inviato dalla GUI e, quindi, la possibilità di inviare ulteriori dati; nel caso in cui non si possegga il *token*, ogni invio di bit sulla linea seriale è disabilitato.
  Alla ricezione del *token*, la transizione nel nuovo stato è discriminata dal segnale ***VALUTA_PRESA***: se attivo, quindi se il giocatore ha già lanciato la sua carta e di conseguenza l'FPGA ha precedentemente lanciato la sua carta, la *Control Unit* si porta nello stato di invio del risultato, asserendo il segnale ***INVIA_RISULTATO*** attivo; se inattivo, l'FPGA non ha ancora lanciato la sua carta perciò il *Controller* si porta nello stato di decisione della carta, asserendo, questa volta, il segnale ***DECIDI_CARTA***;
- *S_INVIA_RISULTATO:* questo stato rappresenta l'ultimo step di ogni turno, in cui, tramite la pressione del pulsante di invio sulla board, si invia il risultato della presa attuale alla GUI.
  Successivamente, la *Control Unit* commuta nello *stato di ricezione della mano*, in cui riceve la nuova carta che viene memorizzata all'interno dei registri appositi, asserendo il segnale ***NUOVO_TURNO*** in modo da resettare i vari registri utilizzati nello svolgimento del turno.
  Al termine della partita, all'invio dell'ultima valutazione della presa, quando la variabile interna ***num_turni*** raggiunge il valore 20, il *Controller* commuta nello *stato di decretazione del vincitore della partita*;

- *S_DECRETA_VINCITORE:* stato finale della partita in cui la *Control Unit* segnala al *Datapath* l'invio del punteggio complessivo della partita.
  Alla ricezione del segnale di reset globale, il *Controller* ritorna nello *stato di IDLE*.

Di seguito è mostrato il codice relativo all'automa appena descritto:

```
case s_current is
	-- STATO 0
	when S_IDLE =>
		DECIDI_CARTA <= '0';
		INVIA_RISULTATO <= '0';
		NUOVO_TURNO <= '0';
		PENULTIMO_TURNO <= '0';

		if(MANO_RICEVUTA = '1') then
			s_current <= S_MANO_RICEVUTA;
			DECIDI_CARTA <= '0';
			INVIA_RISULTATO <= '0';
			NUOVO_TURNO <= '1';
			numTurni := numTurni + 1;
		else
			s_current <= S_IDLE;
		end if;

	-- STATO 1
	when S_MANO_RICEVUTA =>
		if(TOKEN_CPU = '1') then
			DECIDI_CARTA <= '1';
			INVIA_RISULTATO <= '0';
			NUOVO_TURNO <= '0';
			s_current <= S_DECIDI_LANCIA_CARTA;

			if(numTurni = 17) then
				PENULTIMO_TURNO <= '1';
			end if;
		else
			s_current <= S_MANO_RICEVUTA;
		end if;

	-- STATO 2
	when S_DECIDI_LANCIA_CARTA =>
		if(TOKEN_CPU = '0') then
			s_current <= S_ASPETTO_TOKEN;
			DECIDI_CARTA <= '0';
			INVIA_RISULTATO <= '0';
			NUOVO_TURNO <= '0';
```

```
        elsif(TOKEN_CPU = '1' AND VALUTA_PRESA = '1') then
              s_current <= S_INVIA_RISULTATO;
              DECIDI_CARTA <= '0';
              INVIA_RISULTATO <= '1';
              NUOVO_TURNO <= '0';
        else
              s_current <= S_DECIDI_LANCIA_CARTA;
        end if;

  -- STATO 3
  when S_ASPETTO_TOKEN =>
        if(TOKEN_CPU = '1' AND VALUTA_PRESA = '1') then
              s_current <= S_INVIA_RISULTATO;
              DECIDI_CARTA <= '0';
              INVIA_RISULTATO <= '1';
              NUOVO_TURNO <= '0';
        elsif(TOKEN_CPU = '1' AND VALUTA_PRESA = '0') then
              s_current <= S_DECIDI_LANCIA_CARTA;
              DECIDI_CARTA <= '1';
              INVIA_RISULTATO <= '0';
              NUOVO_TURNO <= '0';
        else
              s_current <= S_ASPETTO_TOKEN;
              DECIDI_CARTA <= '0';
              INVIA_RISULTATO <= '0';
              NUOVO_TURNO <= '0';
        end if;

  -- STATO 4
  when S_INVIA_RISULTATO =>
        if(VALUTA_PRESA = '0' AND TASTO_PIGIATO = '1') then
              s_current <= S_DELAY;
              DECIDI_CARTA <= '0';
              INVIA_RISULTATO <= '0';
              NUOVO_TURNO <= '1';
        else
              s_current <= S_INVIA_RISULTATO;
        end if;

  -- STATO DI TRANSIZIONE
  when S_DELAY =>
        if(numTurni = 20) then
              s_current <= S_DECRETA_VINCITORE;
              DECIDI_CARTA <= '0';
              INVIA_RISULTATO <= '1';
              NUOVO_TURNO <= '0';
```

```
        else
                s_current <= S_MANO_RICEVUTA;
                DECIDI_CARTA <= '0';
                INVIA_RISULTATO <= '0';
                NUOVO_TURNO <= '1';
        end if;

    -- STATO 5
    when S_DECRETA_VINCITORE =>
        if(RESET = '1') then
                s_current <= S_IDLE;
                DECIDI_CARTA <= '0';
                INVIA_RISULTATO <= '0';
                NUOVO_TURNO <= '0';
                PENULTIMO_TURNO <= '0';
        else
                s_current <= S_DECRETA_VINCITORE;
        end if;

    end case;
```

Capitolo 6. Graphic User Interface: il lato Java

La GUI si compone di un programma Java che utilizza classi relative al progetto in sé e componenti esterni.

6.1 Classi e relazioni tra le classi

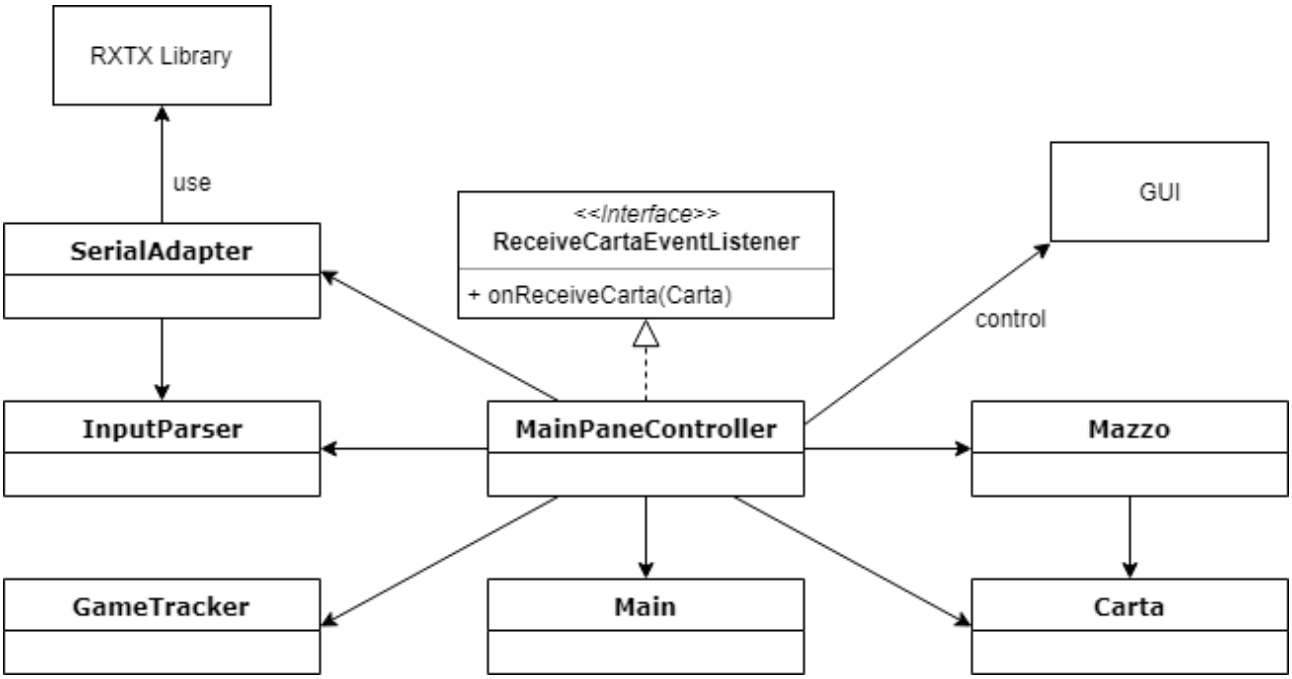

Figura 16.

- *SerialAdapter:* configurazione ed interfacciamento con la porta seriale. Gestisce l'invio dei dati sulla porta COM e scatena un evento ad ogni pacchetto ricevuto. I dettagli della configurazione sono gli stessi del lato VHDL
  Metodi principali:

```
private void readSerial();
private void writeToSerialPort(byte out);
```

- *InputParser:* trasformazione dei pacchetti in informazioni utili per il programma e viceversa; genera i byte delle *carte* e dei *token* da inviare nei pacchetti attraverso il *SerialAdapter*;

Metodi principali:

```
public void parseFrame(byte frame);

public byte fromCartaToByte(Carta c);
```

- *GameTracker:* traccia lo stato del gioco, contiene informazioni per la GUI (chi detiene il turno, chi ha effettuato la presa della mano), genera il *token* in base allo stato del gioco;
  Metodi principali:

```
public byte getToken();
```

- *MainPaneController:* motore della GUI, incapsula tutti i metodi e le unità grafiche presenti nell'interfaccia, gestisce gli eventi scaturiti dall'*utente* e coordina tutte le altre classi per il funzionamento generale del sistema.
  Metodi principali:

```
public void onReceiveCarta(Carta c);

protected void onClickPlayCard(ActionEvent e);
```

- *Main:* classe di base del progetto che carica l'interfaccia grafica e ne delega il controllo al *Controller*;
- *Mazzo:* gestione delle funzioni del mazzo di carte: creazione del mazzo, gestione delle mani dei giocatori, distribuzione delle carte.
  Metodi principali:

```
private ArrayList<Carta> creaMazzo();

public Image getImageFromCarta(Carta c);
```

- *Carta:* contenuto informativo di ogni *carta* da gioco, compresa la sua immagine da visualizzare a schermo.

6.2 Pattern Observer

Il pattern *Observer* (*Publish-Subscriber*) viene utilizzato nei contesti in cui un oggetto richiede che altri oggetti aggiornino il proprio stato interno, notificandone il cambiamento.

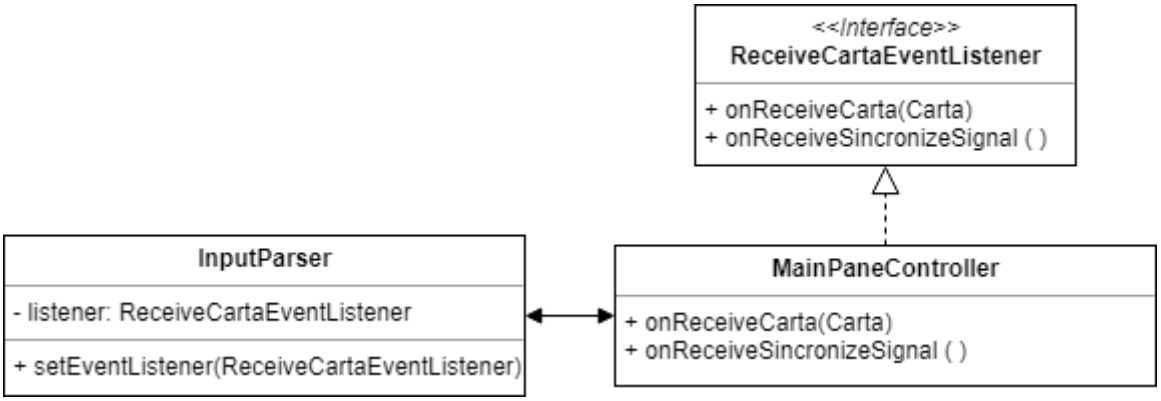

Figura 17. Pattern Observer

In questa applicazione, il *Controller* della GUI crea la classe *InputParser* e successivamente chiama il metodo *setEventListener* passando come argomento sé stesso (*this*). Nel momento in cui l'*InputParser* riceve una carta dalla porta seriale (evento notificato dal *SerialAdapter*), esso chiama il metodo *onReceiveCarta* dopo aver elaborato il frame ricevuto in un oggetto di tipo *Carta*.

La presenza dell'interfaccia ReceiveCartaEventListener permette di rendere il codice riutilizzabile ed espandibile con il solo cambiamento dell'implementazione dei metodi offerti da tale interfaccia.

6.2.1 Implementazione del pattern Observer

Il costruttore della classe *MainPaneController* crea un nuovo *InputParser*, a cui viene passato come parametro il *listener*.

```
public MainPaneController() {
      [...]
      this.parser = new InputParser();
      this.parser.setEventListener(this);
}
```

Alla ricezione di una nuova carta, la classe *InputParser* chiama il metodo *onReceiveCarta* del *listener*, passando come parametro la carta appena ricevuta.

```
public void parseFrame(byte frame) {
      [elaborazione sul frame]
      c = new Carta(valoreCarta, seme);
      c.setBriscola(briscola);
      if (this.listener!=null) listener.onReceiveCarta(c);
}
```

Questo *pattern* è sostanzialmente utilizzato per la creazione di un *evento custom*, creato ad hoc per il problema specifico qualora quelli pre-esistenti non siano adatti allo scopo che si vuole raggiungere.

6.3 Strumenti aggiuntivi: SceneBuilder

Per la creazione delle interfacce grafiche è stato utilizzato il tool *SceneBuilder*, un plugin disponibile per *Eclipse* che ha permesso di disegnare tutte le interfacce grafiche presenti in questo progetto.

Java non supporta, nativamente, l'interfaccia grafica nei suoi programmi: il programmatore deve utilizzare componenti esterni come le librerie *Swing* e *JavaFX*. Entrambe le librerie consentono la realizzazione delle GUI mediante righe di codice, soluzione non molto apprezzata per motivi di semplicità realizzativa.

Lo *SceneBuilder*, basato *JavaFX*, utilizza un file *.xml* (*Briscola.fxml*) in cui memorizza le informazioni relative alle interfacce grafiche e che viene caricato nel *Main* del programma.

Il motore grafico *MainPaneController* utilizza dei *tag* nella dichiarazione degli oggetti grafici e nei metodi che scatenano eventi per permettere allo *SceneBuilder* di referenziarli e legare l'oggetto grafico al suo comportamento al determinarsi di un certo evento. Esempio di *tag*:

Tag: `@FXML`

6.4 Funzionamento

All’avvio della partita viene richiesto il nome del giocatore, il quale verrà poi visualizzato in fase di gioco.

Figura 18. Schermata iniziale gioco

Una volta inserito il nome, ci si troverà nella seconda sezione. In questa fase preliminare è necessario far comunicare direttamente i due componenti per poter instaurare una corretta comunicazione.

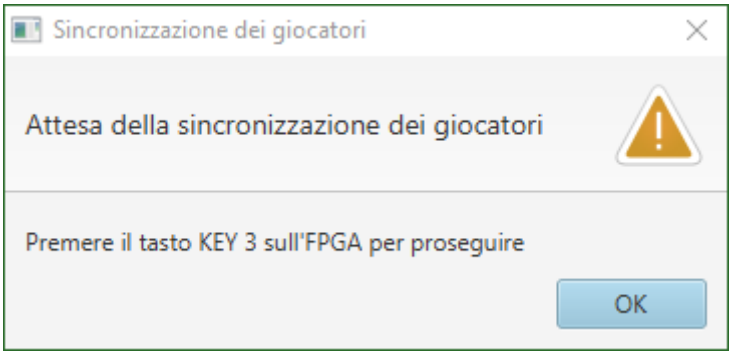

Figura 19. Sincronizzazione sistema

Successivamente, viene richiesto all'utente quale giocatore inizierà la partita.

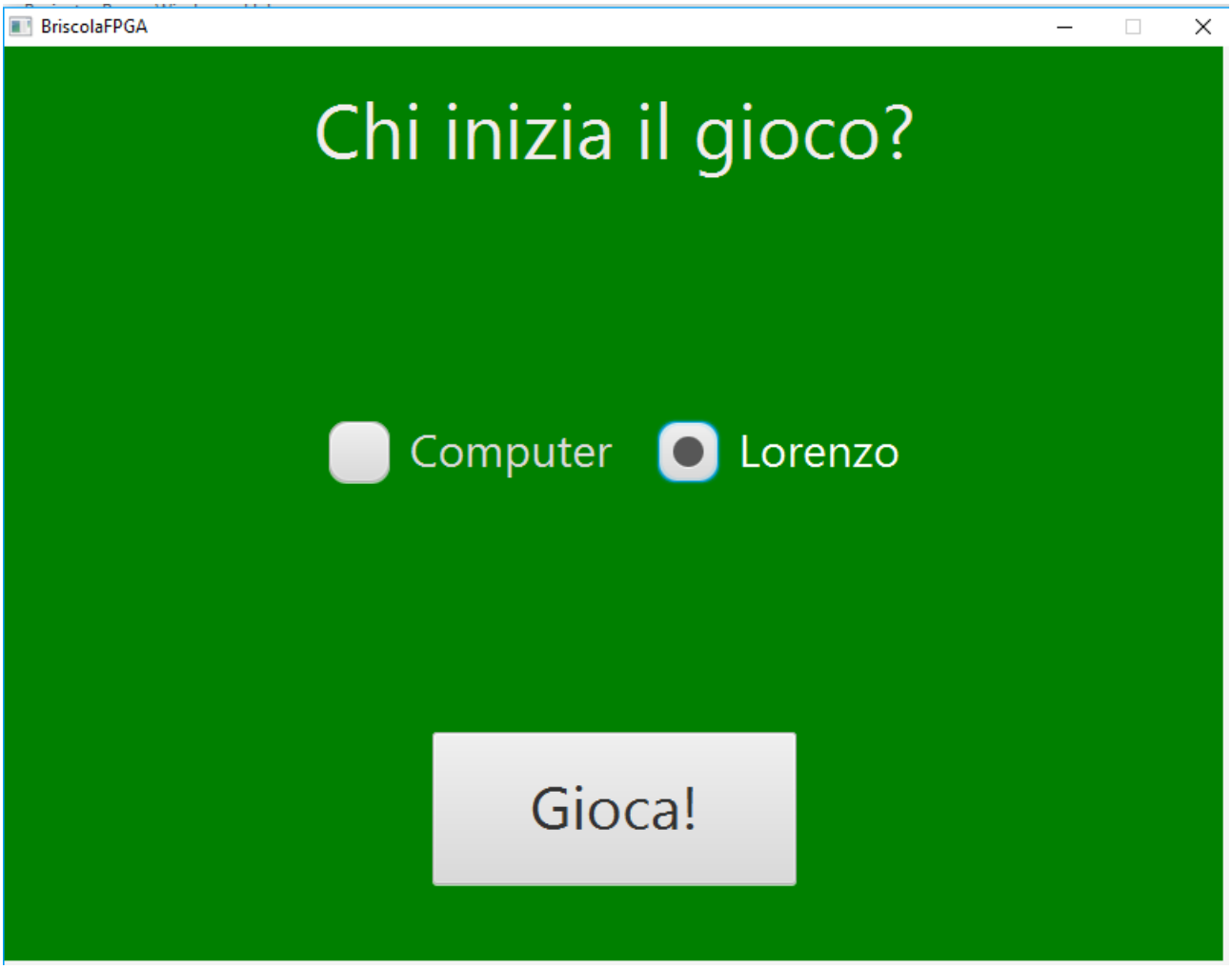

Figura 20. Scelta giocatore

A questo punto, cliccando il pulsante *Gioca!*, ci si troverà nella schermata principale di gioco; contemporaneamente, l'applicazione distribuisce le carte all'FPGA, che è pronta ad aspettare la prossima mossa.

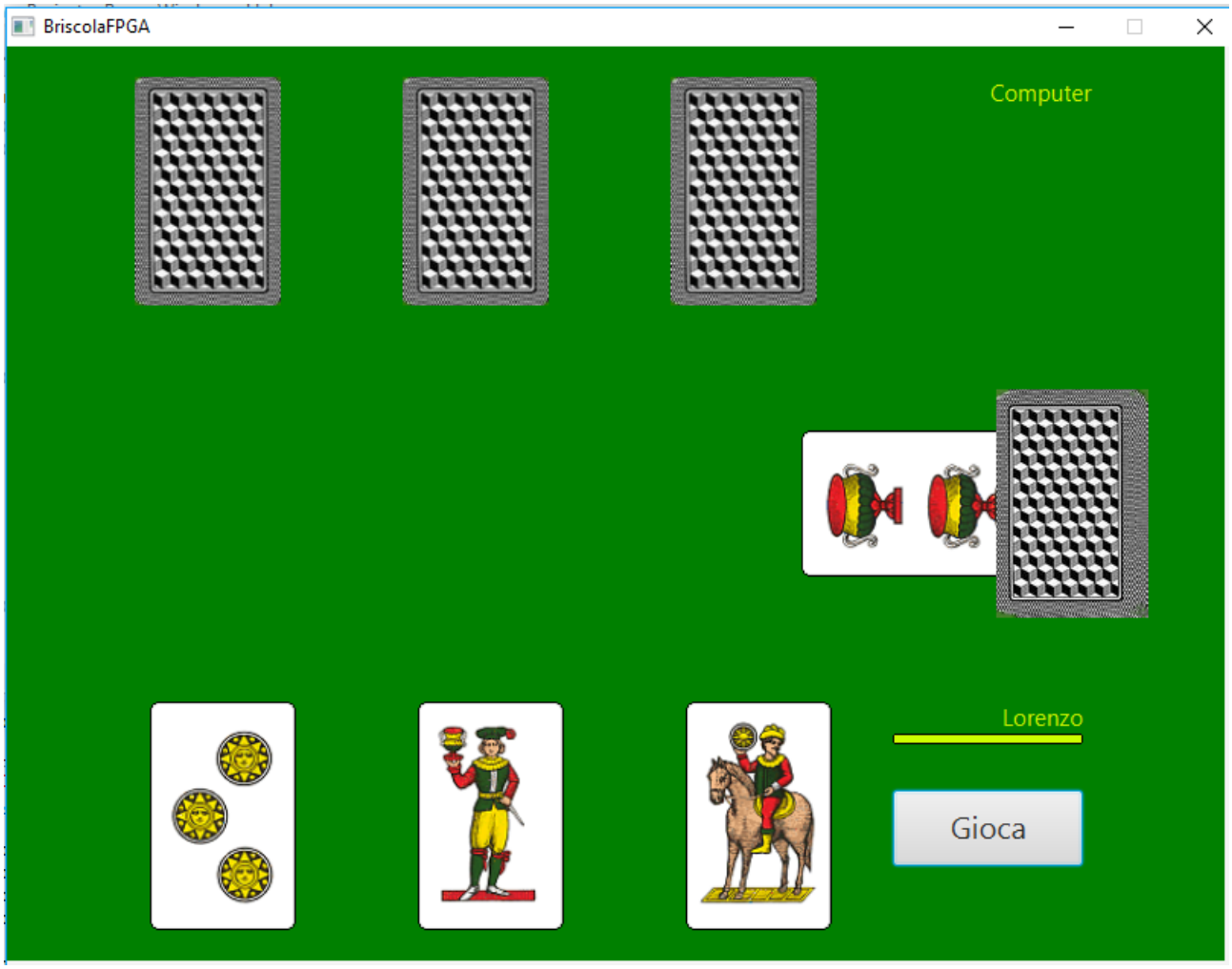

Figura 21. Schermata di gioco

Qui è mostrata la schermata di fine partita:

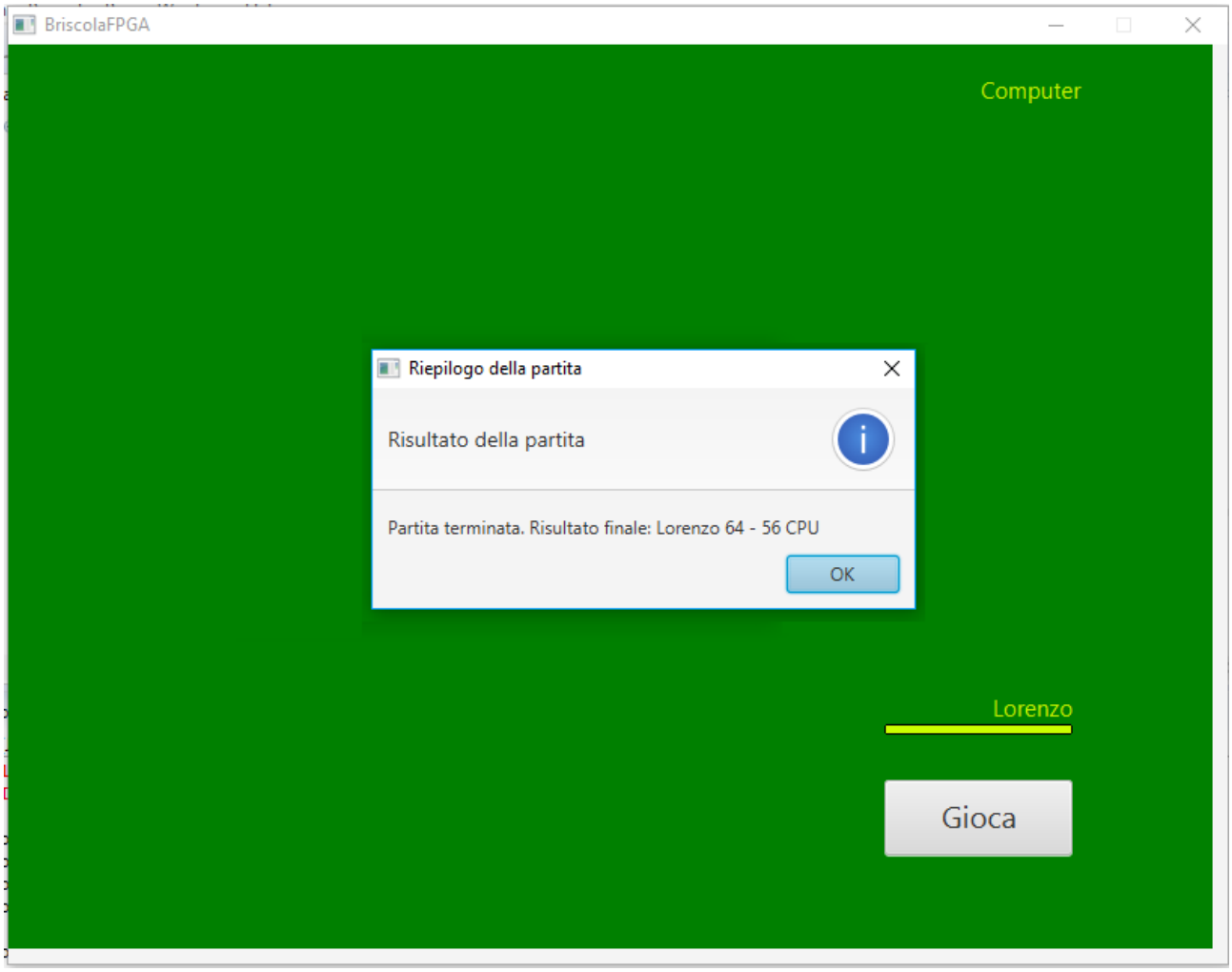

Figura 22. Schermata di fine partita

Capitolo 7. Punto d'incontro: comunicazione seriale

L'utente interagisce con la GUI Java, la quale notifica alla DE1 tutti i cambiamenti effettuati. La comunicazione avviene tramite *protocollo seriale RS232* con interfaccia *UART*.

7.1 Interfaccia UART e protocollo seriale

Il sistema incapsula un'interfaccia *UART* (*Universal Asynchronous Receveir-Tramsitter*) indispensabile per la trasmissione dei bit: il modulo in questione ha il compito fondamentale di convertire il flusso di dati da parallelo a seriale e viceversa per poter essere e ricevuto tramite i pin ***UART_TXD*** e ***UART_RXD*** presenti sulla scheda DE1.

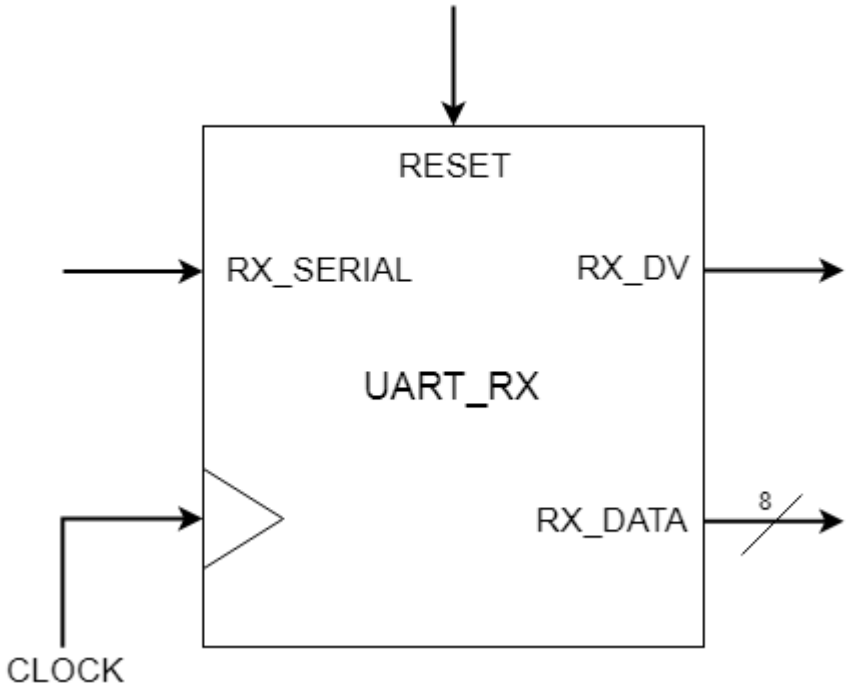

Figura 23. Entità UART_RX

Questa entità è stata progettata come un *automa a stati finiti*:

```
type s_briscola is (S_IDLE, S_RX_START_BIT, S_RX_DATA_BIT, S_RX_STOP_BIT,
                    S_CLEANUP);
```

Il funzionamento dell'automa è di seguito spiegato:

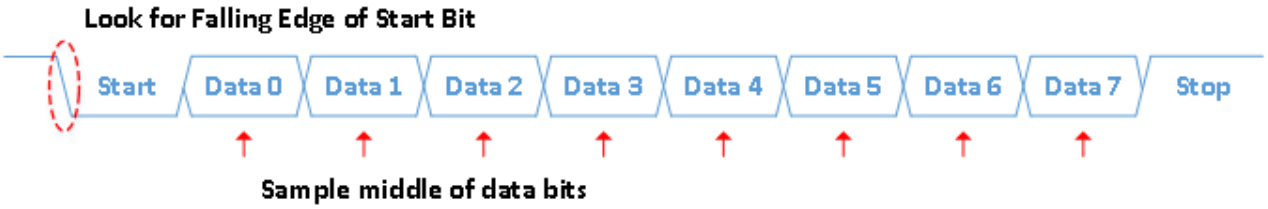

Figura 24. Diagramma temporale ASF UART_RX

L'interfaccia *UART* mostra all'utente alcuni parametri personalizzabili:

- *Baud rate*, è la velocità con cui vengono trasmessi i dati seriali
  Valori possibili:
  9600, standard;
  19200;
  115200;
  valore custom;
- *Numero di bit di dato,* da sette a otto bit;
- *Bit di parità*: utilizzato per il controllo dell'errore, può non essere presente;
- *Numero di bit di stop:*
  Valori possibili:
  assente;
  1;
  2;
- *Controllo di flusso:*
  Valori possibili:
  assente (standard);
  on;
  hardware;

I valori scelti per questo progetto sono, in ordine: (9600, 8, 0, 1, assente).

7.2 Struttura delle carte e dei token

Come già accennato precedentemente, il modulo *UART_RX* invia dati sotto forma di byte: il primo bit serve a discriminare se il dato in questione è

- una *carta*, bit ad 1;
- un *token*, bit a 0.

In presenza di una *carta*, i bit 1, 2, 3, 4 indicano il valore della carta stessa, i bit 5 e 6 il suo *seme* e il 7 bit se essa sia o meno una briscola.

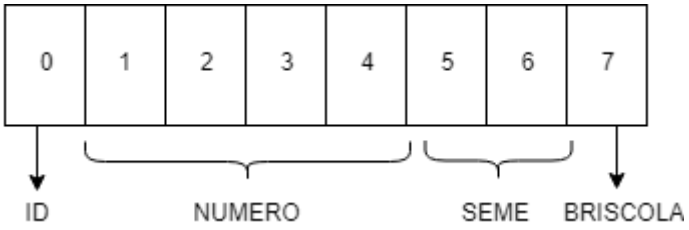

Figura 25. Pacchetto di tipo carta

In presenza di un token, i bit 1, 2, 3 discriminano il tipo di token, i restanti bit servono a definire chi ha preso nella mano corrente:

- bit 1, 2, 3:
  111 -> tocca a chi lo riceve;
  000 -> non tocca a chi lo riceve;
- bit 4, 5, 6, 7:
  0101 -> non si sa ancora chi ha preso;
  1111 -> ha preso l'FPGA;
  0000 -> ha preso il giocatore umano;

Caso particolare è la configurazione dei quattro bit finali 1010: essa indica il *reset_token*, utile al reset dei registri del *Datapath*.

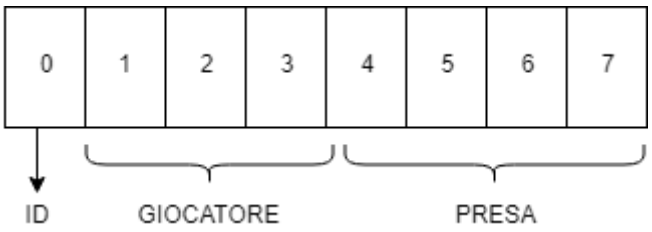

Figura 26. Pacchetto di tipo token

7.3 Fasi della comunicazione

1. L'*utente* sceglie chi sarà il primo giocatore a tirare;
2. L'applicazione JAVA invia all'FPGA le carte della sua *mano*, la *briscola dominante* e il *token*;

Caso FPGA prima:

3. L'FPGA sceglie la carta da giocare, gioca ed invia la *carta scelta* e il *token*;
4. L'*utente* risponde ed invia la *carta scelta* e il *token*;
5. L'FPGA valuta la presa e invia il risultato con il *token* corrispondente;
6. L'applicazione JAVA invia all'FPGA il *token reset* e la nuova carta da inserire nella mano.

Caso utente prima:

3. L'*utente* gioca ed invia una propria *carta* e il *token*;
4. L'FPGA sceglie la *carta* con cui rispondere, la gioca, valuta la presa e invia la *carta* e il *token* corrispondente alla presa;
5. L'applicazione JAVA visualizza la *carta* giocata dall'FPGA;
6. L'applicazione JAVA invia all'FPGA il *token reset* e la nuova carta da inserire nella mano.

7.4 Libreria RXTX

RXTX è una libreria Java che permette di utilizzare la comunicazione seriale e parallela per il JDK.

7.4.1 Parametri di configurazione

```
private static final int DATA_RATE = 9600;
serialPort.setSerialPortParams(DATA_RATE,
                               SerialPort.DATABITS_8,
                               SerialPort.STOPBITS_1,
                               SerialPort.PARITY_NONE);
```

7.4.2 Apertura degli stream e aggiunta degli Event Listeners

```
// open the streams
is = serialPort.getInputStream();
output = serialPort.getOutputStream();
os = new PrintStream(output, true);

// add event listeners
serialPort.addEventListener(this);
serialPort.notifyOnDataAvailable(true);
```

7.4.3 Evento di arrivo dei dati dalla porta seriale

```
public synchronized void serialEvent(SerialPortEvent serialEvent) {
     if (serialEvent.getEventType() ==
                                SerialPortEvent.DATA_AVAILABLE) {
          try {
               readSerial();
          } catch (Exception e) {
               System.err.println(e.toString());
          }
     }
}
```

7.4.4 Lettura dei dati dalla porta seriale

```
private void readSerial() {
     int availableBytes = is.available();
     if (availableBytes > 0) {
          // Read the serial port
          is.read(readBuffer, 0, availableBytes);
     }

     for(int i = 0; i < availableBytes; i++) {
          parser.parseFrame(readBuffer[i]);
     }
}
```

7.4.5 Scrittura dei dati sulla porta seriale

```
public void writeToSerialPort(byte out) {
      System.out.print("\nInvio: ");
      System.out.println(Integer.toBinaryString(out));
      os.write(out);
}
```

Captolo 8. Audio

Per sfruttare al meglio le potenzialità della scheda, abbiamo deciso di inserire un sottofondo musicale che è possibile ascoltare durante la partita. La scelta è ricaduta su un frammento di liscio romagnolo *"Valzer romagnolo"* di Clementino (*IMG Edizioni*) per ricreare un'atmosfera conviviale tipica delle sagre e delle feste di paese.

8.1 Codec stereo WM8731

La DE1 contiene un codec stereo appositamente progettato per la lettura di file mp3, il WM8731.

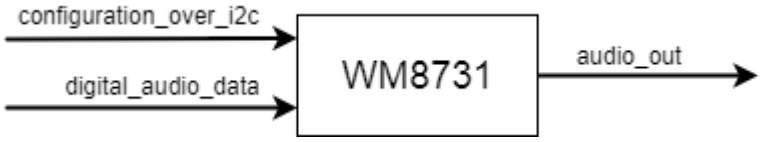

Figura 27. Schema funzionamento codec audio

Come si evince dalla Figura 1, il codec legge dei dati audio digitali sotto forma di stringhe da 16 a 32 bit, campionandole con una frequenza variabile da un minimo di 8KHz ad un massimo di 96 KHz. Può funzionare sia come dispositivo MASTER che come SLAVE. Possiede un'interfaccia che fornisce:

- controlli di volume;
- possibilità di silenziare l'audio;
- possibilità di usare il codec in modalità stereo o mono

il tutto configurabile tramite il protocollo I2C.

La configurazione da noi usata per questo progetto è la seguente:

- codec in SLAVE mode;
- campionamento in USB mode:
    - clock a 12 MHz;
    - frequenza di campionamento di 48kSps;
    - risoluzione audio da 16 bit
- campionamento tramite PCM (*pulse-code modulation*);
- modalità mono

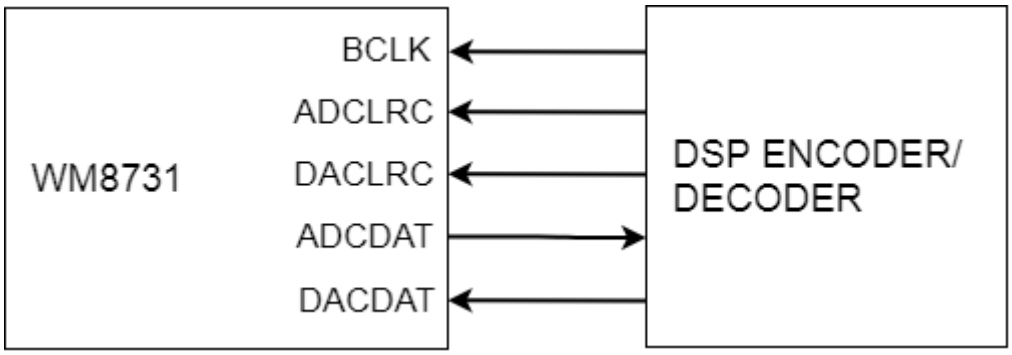

Figura 28. WM8731 in SLAVE mode

Il DSP (*Digital signal processor*), mostrato in figura 2, è un hardware dedicato e ottimizzato presente sulla scheda DE1 che permette di elaborare efficientemente segnali digitali.

Il WM8731, in SLAVE mode, riceve dal DSP il clock a cui tutto il sistema audio lavorerà (*BCLK, Digital Audio Bit Clock*) e i due clock per la sincronizzazione delle operazioni di conversione da analogico a digitale (*ADCLRC, ADC sample rate left/right clock*) e viceversa (*DACLRC, DAC sample rate left/right clock*); dopodichè, il codec stereo invia i dati audio da convertire (*pin ADCDAT, ADC digital audio data output*) e ne riceve la conversione (*pin DACDAT, DAC digital audio data output*).

8.2 Protocollo I^2C

L'I^2C (*Inter Integrated Circuit*), sviluppato nel 1982 dalla Philips, ma divenuto uno standard largamente diffuso solo nel 1992, è il protocollo che abbiamo scelto per la configurazione del codec audio.

Trattasi di un protocollo seriale che permette l'interfacciamento tra uno o più device *master* con un numero illimitato di altri device *slave*. Sono presenti solo due bus:

- SCL: *serial clock*, utilizzato per definire un clock unico per la connessione;
- SDA: *serial data*, per i dati effettivi;

entrambi i bus sono bidirezionali.

I device *master* generano il clock, danno inizio e fermano la connessione con i vari altri componenti e inviano i vari comandi permessi dal protocollo.

Una trasmissione dati base avviene in questo modo:

1. in principio, i due bus sono posti a livello alto;
2. il *master* genera una *Start condition* (SCL = 1, SDA = 0) seguito dall'indirizzo dello *slave* interessato: se il LSB è posto a 0, il master intende scrivere (*write mode*);
3. attende un *ack* da parte del device scelto;
4. in caso di *ack* positivo, invia un byte di dato;
5. attende un *ack* da parte del device scelto;
6. i punti 5 e 6 si ripetono fintanto che non venga generata una *Stop condition* (SCL = 1, SDA = 1).

Di seguito è mostrato tramite l'uso di diagrammi temporali quanto appena spiegato.

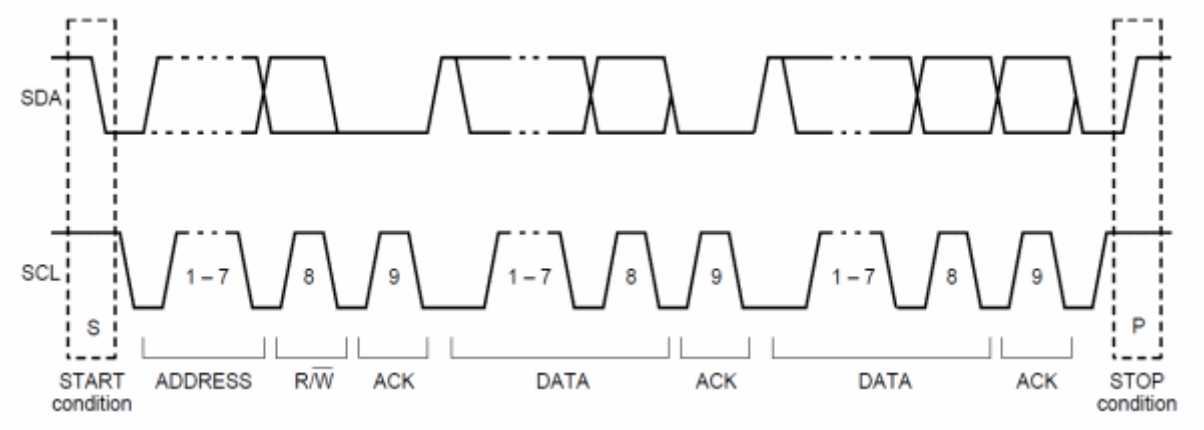

Figura 29. Funzionamento del protocollo I²C

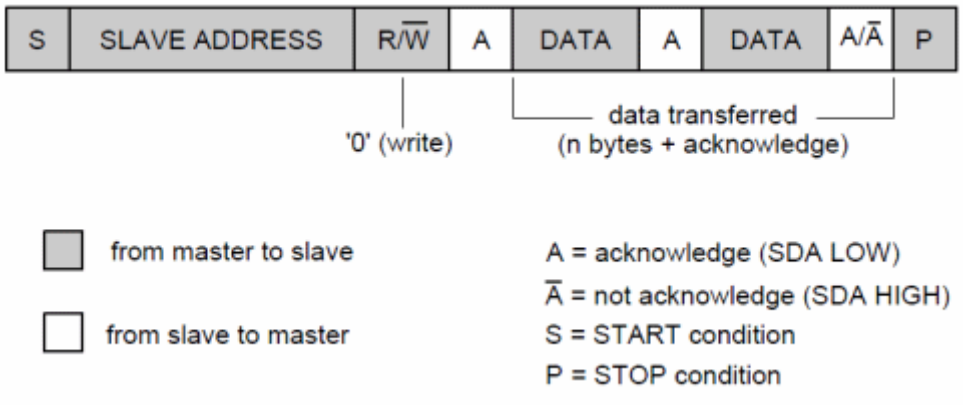

Figura 30. Esempio di comunicazione I²C

Ai fini dell'implementazione di *BriscolaDE1*, è stata definita un'*entità* apposita che implementasse il protocollo scelto.

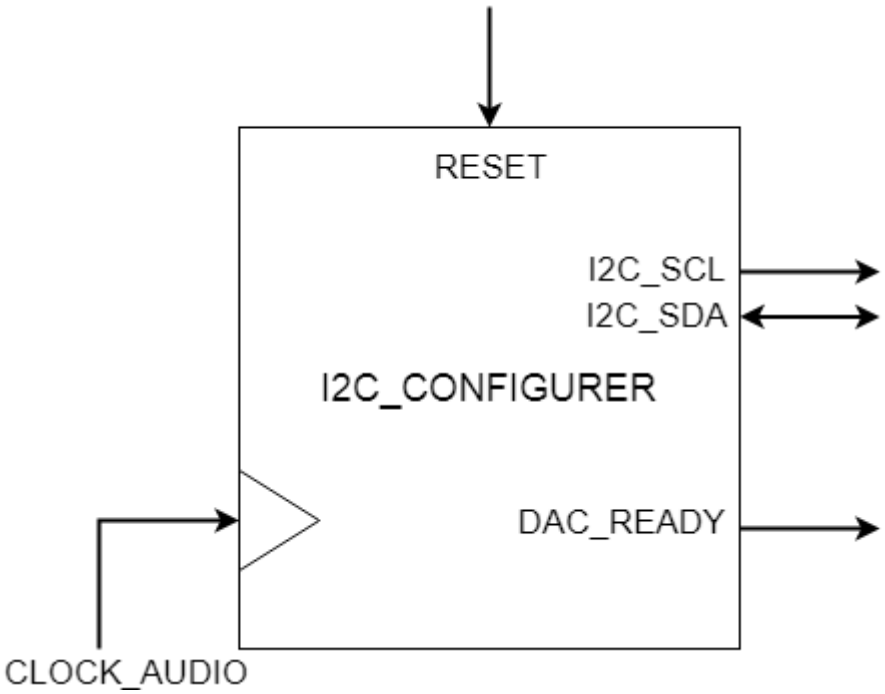

Figura 31. Entità I^2C

Il suo comportamento è stato definito utilizzando i seguenti *process*:

- `I2CClockProcess : process(CLOCK_12, RESET)`

  si occupa di generare, tramite l'utilizzo di un prescaler, i clock utili al protocollo e le transizioni tra i dati;

```
elsif(rising_edge(CLOCK_12)) then
	if(clk_prs < I2C_PRESCALER) then
		clk_prs <= clk_prs + 1;
	else
		clk_prs <= 0;
	end if;

	-- 50 % duty cylce clock for i2c
	if(clk_prs < I2C_PRESCALER/2) then
		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;
   end if;
```

- `FSMI2CProcess : process(CLOCK_12, RESET)`

  definisce un automa a stati finiti che scandisce il regolare funzionamento del protocollo; a grandi linee, può essere visto nel seguente modo:

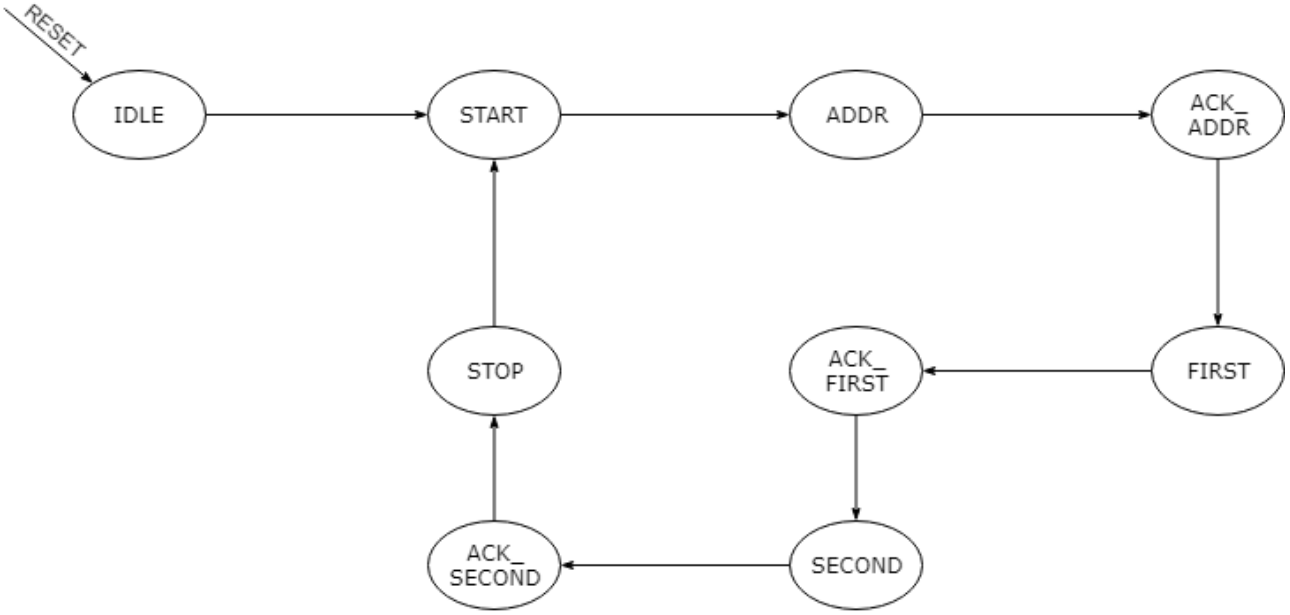

Figura 32. ASF dell'I2C_codec

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

- `InitProcess : process(CLOCK_12, RESET)`

  si occupa della configurazione del codec WM8731

```
        case init_counter is
        when 0 =>
              -- reset
```

```
        i2c_data(15 downto 9) <= "0001111";
        i2c_data(8 downto 0) <= "000000000";
        i2c_send_flag <= '1';

when 1 =>
        -- active interface
        i2c_data(15 downto 9) <= "0001001";
        i2c_data(8 downto 0) <= "111111111";
        i2c_send_flag <= '1';
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';
when 3 =>
        -- Digital Interface: DSP, 16 bit, slave mode
        i2c_data(15 downto 9) <= "0000111";
        i2c_data(8 downto 0) <= "000010011";
        i2c_send_flag <= '1';
when 4 =>
        -- headphone volume
        i2c_data(15 downto 9) <= "0000010";
        i2c_data(8 downto 0) <= "101111001";
        i2c_send_flag <= '1';
when 5 =>
        -- USB mode
        i2c_data(15 downto 9) <= "0001000";
        i2c_data(8 downto 0) <= "000000001";
        i2c_send_flag <= '1';
when 6 =>
        -- enable DAC to LINOUT
        i2c_data(15 downto 9) <= "0000100";
        i2c_data(8 downto 0) <= "000010010";
        i2c_send_flag <= '1';
when 7 =>
        -- remove mute DAC
        i2c_data(15 downto 9) <= "0000101";
        i2c_data(8 downto 0) <= "000000000";
        i2c_send_flag <= '1';
        init <= '1';
        dac_ready <= '1';
when others => … -- reset data and flag
end case;
```

8.4 Interfaccia PCM

In base a quanto mostrato fino a questo punto, l'entità di top-level che si occuperà della configurazione del codec audio WM8731 (tramite l'utilizzo dell'I2C_CONFIGURER) e della riproduzione dell'audio sarà la seguente:

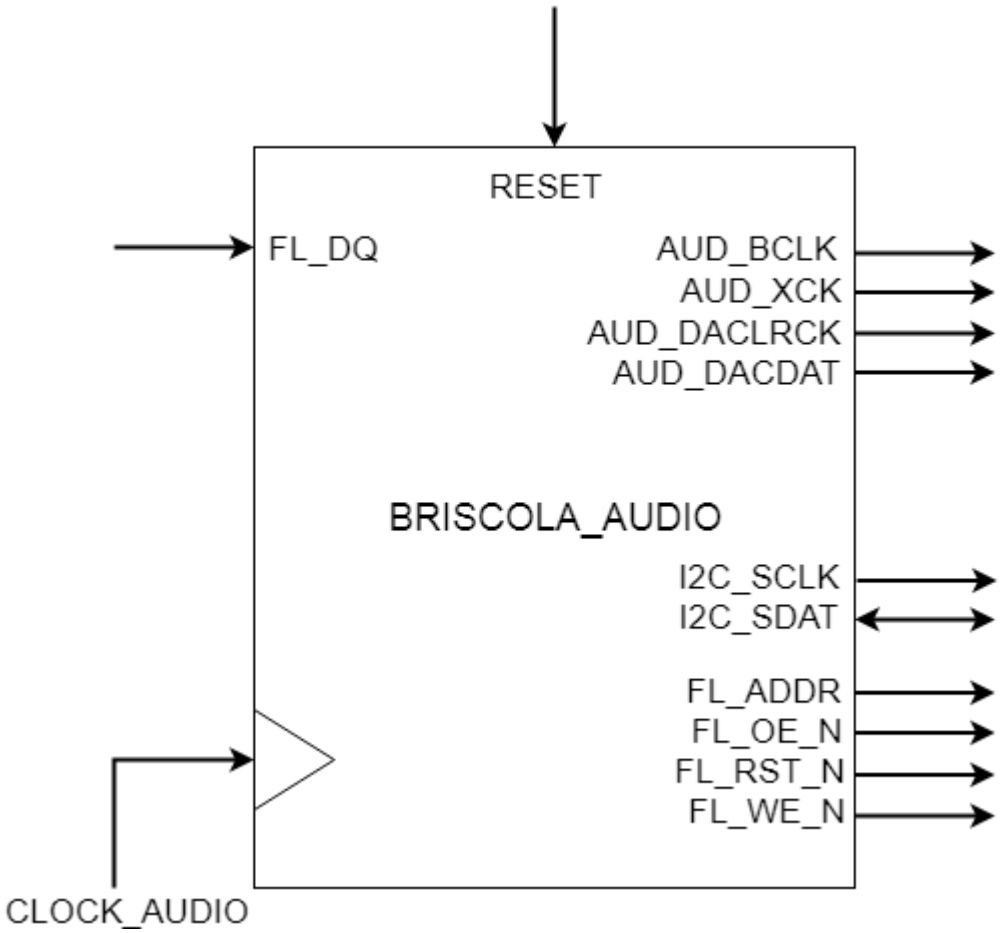

Figura 33. Entità PCM

Per semplicità di progettazione, abbiamo deciso di suddividere i due compiti del componente in due *process* separati:

- `AudioGenProcess : process(CLOCK_AUDIO, RESET)`

  genera l'audio inviato al codec WM8731: si occupa di ricevere i dati audio e, una volta campionati, li invia al sistema audio della scheda DE1

```
if(DAC_READY = '1') then
      -- 48k sample rate
      if(audio_prescaler < AUDIO_PRESCALER_MAX_VALUE) then
            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;
      end if;
```

- `ReadAudioProcess : process(CLOCK_AUDIO, RESET)`

  legge il file audio 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;
      end if;
```

Per la memorizzazione del file, abbiamo optato per l'utilizzo delle memorie integrate alla scheda DE1, nel caso specifico abbiamo sfruttato la semplicità d'uso della memoria FLASH da 4Mb presente. Il file audio è stato salvato utilizzando il System Control Panel della DE1.

Figura 34. System Control Panel DE1

Il file audio "*liscio.wav*", con risoluzione a 16 bit e campionamento a 48kHz, ha una dimensione di 1.58MB, per una lunghezza di 17 secondi. Ciò che sarà possibile ascoltare consiste in un loop del suddetto.

Capitolo 9. Fonti e bibliografia

1. UART_RX: *https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html*
2. AUDIO_CODEC: *https://www.youtube.com/watch?v=zzIi7ErWhAA*
3. Figura 24: *https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html*
4. Figura 29: *https://i2c.info/i2c-bus-specification*
5. Figura 30: *https://i2c.info/i2c-bus-specification*