# 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 | Alessandro Paoletti | Alessandro Morabito |
| 0000901917 | 0000901759 | 0000907048 |

Anno Accademico: 2019/2020

# Sommario (da inserire solo alla fine)

[Capitolo 1. La Briscola 2](#_Toc18877272)

[Capitolo 2. Architettura del sistema 2](#_Toc18877273)

[Capitolo 3. Datapath 2](#_Toc18877274)

[Capitolo 4. Control Unit 2](#_Toc18877275)

[Capitolo 5. Graphic User Interface: il lato Java 2](#_Toc18877276)

[Capitolo 6. Punto d’incontro: comunicazione seriale 2](#_Toc18877277)

[Capitolo 7. Audio Errore. Il segnalibro non è definito.](#_Toc18877278)

[7.1 Codec stereo WM8731 **Errore. Il segnalibro non è definito.**](#_Toc18877279)

[7.2 Protocollo I2C **Errore. Il segnalibro non è definito.**](#_Toc18877280)

[7.3 Interfaccia PCM **Errore. Il segnalibro non è definito.**](#_Toc18877281)

[7.4 Entità di top-level **Errore. Il segnalibro non è definito.**](#_Toc18877282)

[Capitolo 8. Fonti 2](#_Toc18877283)

# 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” (lanciare per primi alla successiva mano ed, eventualmente, ottenere dei punti se le carte prese non sono dei lisci) solo se la propria carta messa sul tavolo ha lo stesso seme, punteggio maggiore o, in caso di parità, numero maggiore. 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 |
| 1. Mischiare le carte del mazzo | JAVA |
| 1. Inviare le tre carte iniziali al giocatore virtuale | JAVA |
| 1. Visualizzare carte giocatore reale, virtuale e briscola | JAVA |
| for(i = 0; i < 20; i++) {  if(token == CPU) { |  |
| * 1. Scegliere la carta da giocare | VHDL |
| * 1. Inviare la carta giocata | VHDL |
| * 1. Attendere il *token* | VHDL |
| * 1. Decidere chi ha preso | VHDL |
| * 1. Assegnare i punti della presa | VHDL |
| else { |  |
| 1. Attendere il proprio turno (*token*) | VHDL |
| 1. Scegliere la carta da giocare | VHDL |
| 1. Inviare la carta scelta | VHDL |
| 1. Decidere chi ha preso | VHDL |
| 1. 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;
* *fase 2*: il giocatore gioca per primo;

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:

# Capitolo 3. Architettura del sistema

## 3.1 Entità di top-level

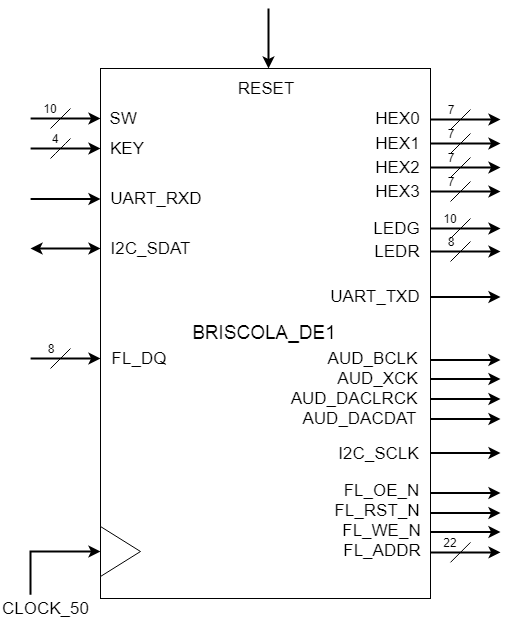
L’*entità di top-level* rappresenta l’entità più astratta di tutto il sistema, la quale si interfaccerà direttamente con la scheda DE1. È stata definita nel file *Briscola.vhd*.

Figura 1. 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 I2C 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

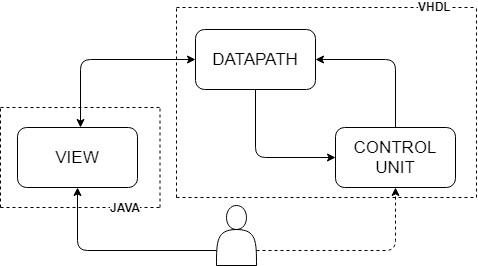
Il *sistema* è basato sulla seguente architettura:

Figura 2. 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 (capitolo );
* *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 (capitolo );

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, capitolo ) 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**

-- AUDIO CONSTANTS –

constant LAST\_FLASH\_ADDR : positive := 1661635;

constant AUDIO\_PRESCALER\_MAX : positive := 250;

constant I2C\_PRESCALER : positive := 60;

### **3.3.3 briscola\_datapath\_package**

-- audio costant

function decidiCarta(mano : mano\_cpu) return integer;

-- audio costant

function valutaPresa(carta1 : carta; carta2 : carta) return boolean;

### **3.3.4 briscola\_utility\_package**

-- dato un numero, genera il suo codice equivalente per un display a 7 segmenti

function numberTo7SegmentDisplay(numero : integer) return std\_logic\_vector;

-- dato un carattere, genera il suo codice equivalente per il display a 7 segmenti

function digitTo7SegmentDisplay(carattere : character) return std\_logic\_vector;

-- dato il numero della carta, restituisce il suo 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;

La usiamo perché utilizzando i vettori con indici 0 to 7 il bit nella posizione 0 è l’MSB mentre nel modulo di trasmissione e ricezione dalla porta seriale vengono usati vettori con indici 7 downto 0, in cui il bit nela posizione 0 è il LSB.

-- controlla se un vettore è pieno di zeri

function vectorIsNotZero(vector : std\_logic\_vector) return boolean;

-- data la carta, la trasforma nel corrispondente byte

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 vero se è presente un carico nella mano, falso altrimenti

function isCarico(mano : mano\_cpu) return boolean;

-- restituisce vero se esistono un liscio e un carico con lo stesso seme, falso 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 in 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 vero se nella mano ci sono più carichi dello stesso seme, falso -- 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 vero se nella mano sono presenti degli assi, falso 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 vero se la briscola presente è con punti (sia figure che

-- carichi), falso 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;

-- FUNZIONE di TOP\_LEVEL –

-- restituisce il numero di carichi in mano

function getNumeroCarichi(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP\_LEVEL --

-- restituisce il numero di briscole in mano

function getNumeroBriscole(mano : mano\_cpu) return integer;

-- FUNZIONE di TOP\_LEVEL --

-- 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**

--

function decidiCartaFase2(mano : mano\_cpu; cartaTerra : carta) return integer;

--

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

--

function decidiPenultimo(mano : mano\_cpu; briscola\_partita : carta) return integer;

--

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, svolgendo i compiti di:

* ricezione, elaborazione e memorizzazione dei dati ricevuti dalla parte software (Java);
* individuazione della carta da giocare, secondo il set di regole inserito nei package del progetto;
* calcolo del risultato delle prese e generazione dei token corrispondenti;
* invio dei risultati delle prese e delle carte giocate

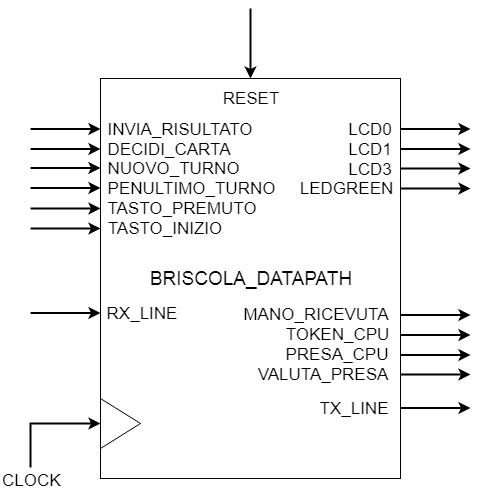
Di seguito è mostrato lo schema a blocchi dell’entità relativa al datapath:

Figura 3. Entità Briscola\_Datapath

Il datapath a sua volta si suddivide in quattro moduli comunicanti tra loro attraverso la control unit, che svolgono indipendentemente ciascuna delle quattro funzioni descritte in precedenza.

## 4.1 Tipi di dato

All’interno del briscola\_package è presente la definizione dei tipi di dato utilizzati per definire le entità principali utilizzate. La realizzazione delle strutture dati è trattata nel capitolo .

## 4.2 Modulo di ricezione delle carte

Questo modulo abilita la scheda 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, che indica l’inizio della computazione della scelta della carta.

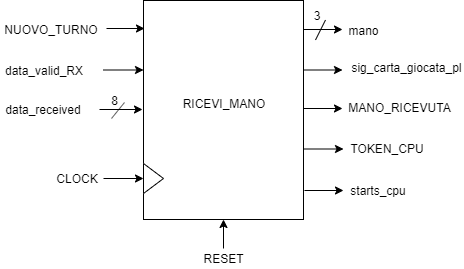
Di seguito riportiamo lo schema a blocchi di questo modulo:

Figura 4. Modulo di ricezione carte

Il componente riceve in ingresso i seguenti segnali:

* ***NUOVO\_TURNO:*** proveniente dalla control unit, inizializza i segnali interni per l’inizio di un nuovo turno della partita;
* ***data\_valid\_RX:*** proveniente dal modulo UART\_RX, segnala al ricevitore che il pacchetto ricevuto sulla linea seriale è completo e pronto per essere letto;
* ***data\_received:*** proveniente dal modulo UART\_RX, contiene il dato di 8 bit ricevuto dalla porta seriale

Fornisce in uscita i seguenti segnali:

* ***mano:*** array di 3 carte, costituiscono la mano della CPU da cui scegliere la carta da giocare;
* ***sig\_carta\_giocata\_pl:*** contiene la carta lanciata dal giocatore mediante la view; viene utilizzata dagli altri moduli per selezionare la carta corretta da lanciare e per la valutazione della 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 vengono riportati frammenti di codice significativo per descrivere il 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*** si trova 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 fatta per i *token*, con un’elaborazione più semplice vista la loro configurazione fissa, andando a resettare i dati del turno nel caso di arrivo del *reset token*.

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 vanno a considerare i bit relativi al detentore effettivo del *token*, attivando il corrispondente segnale in uscita 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

Questo modulo serve a 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 View.

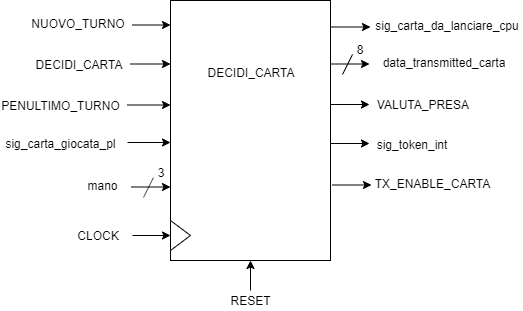
Di seguito lo schema a blocchi del modulo in questione:

Figura 5. 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****:* proveniente dalla control unit, permette al modulo di selezionare la carta da lanciare;
* ***PENULTIMO\_TURNO***: proveniente 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***

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***: la carta scelta per essere giocata, trasformata nel formato a 8 bit da inviare alla View;
* ***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 View

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 va a resettare 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 scegliere la strategia opportuna, con regole diverse da quelle della partita pregressa.

if(DECIDI\_CARTA = '1') then

if(PENULTIMO\_TURNO = '1') then

indice := decidiPenultimo(mano, briscola\_partita);

Le regole si suddividono in fase 1 e fase 2; la condizione che le differenzia è espressa nel registro che contiene la carta lanciata dal giocatore: se esso contiene ancora il valore della carta reset, il giocatore non ha ancora lanciato perciò si sceglierà la carta seguendo il set di regole della fase 1. Qualora il giocatore avesse già lanciato una carta, si sceglierà la carta seguendo il set di regole della fase 2.  
La carta viene individuata attraverso la sua posizione all’interno della mano della CPU.

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 ed abilitato il segnale ***VALUTA\_PRESA*** se il giocatore ha già lanciato la sua carta. La funzione ***fromCartaToByte***, presente nel package ***briscola\_utility\_package***, permette di tradurre istantaneamente 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

Questo 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.

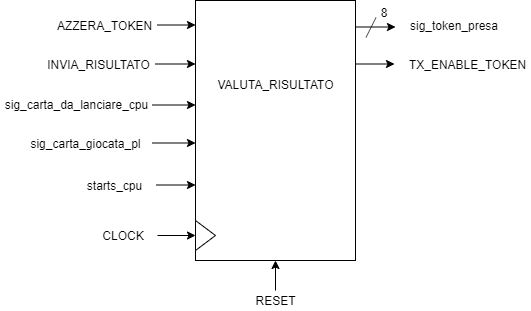
Di seguito lo schema a blocchi del modulo in questione:

Figura 6. 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***: proveniente dalla control unit, permette 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 capire chi comanda la presa.

Fornisce in output i seguenti segnali:

* ***sig\_token\_presa***: token parziale che viene riempito 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 inviato alla View.

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 controlla il valore del segnale ***starts\_cpu*** (un booleano); se la funzione di valutazione della presa valutaPresa indica che la CPU ha preso (***risultato*** = true), allora si sommano i punti delle carte giocate da entrambi i giocatori nella variabile accumulatrice di punti e si predispone il token con la configurazione corrispondente; se viceversa ha preso il giocatore (***risultato*** = false), si sommano i punti delle carte giocate da entrambi i giocatori nella variabile dei punti del giocatore.

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 valore del segnale ***starts\_cpu*** sia falso – all’inizio del turno il giocatore 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";

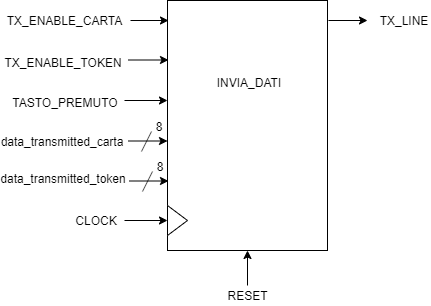
end if;

## 3.5 Modulo di trasmissione dei dati

Questo modulo viene utilizzato per l’invio diretto dei dati sulla porta seriale. A differenza della ricezione, in cui viene usato un modulo esterno che fornisce il flag di dato completo e il byte ricevuto, in questo modulo l’invio dei bit sulla porta seriale viene effettuato direttamente dal modulo stesso, occupandosi della composizione del pacchetto e della sincronizzazione con il baud rate.

Di seguito lo schema a blocchi del modulo in questione:

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

  
Figura 7: Schema a blocchi del modulo di trasmissione dei dati

* ***TX\_ENABLE\_CARTA***, ***TX\_ENABLE\_TOKEN***: segnali di enable che indicano la validità dei byte di carta e 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 di carta e token pronti per essere inviati sulla linea seriale.

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 realizzazione dell’invio dei dati sono necessarie due variabili fondamentali:   
- 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; al valore 0 del bit\_number viene inviato lo start bit, al valore 9 viene inviato lo stop bit, mentre nel range 1 e 8 vengono inviati i bit dei dati da inviare. Il flag ***byte\_not\_sent*** permette di inviare una carta alla volta ad ogni pressione del tasto; senza questo flag la trasmissione dei dati sarebbe continua, inviando 9600 bit/s per ogni secondo che 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

TX\_LINE <= data\_transmitted\_carta(bit\_number-1);   
 --8 data bits 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

TX\_LINE <= data\_transmitted\_token(bit\_number\_tk-1);  
 --8 data bits  
 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;

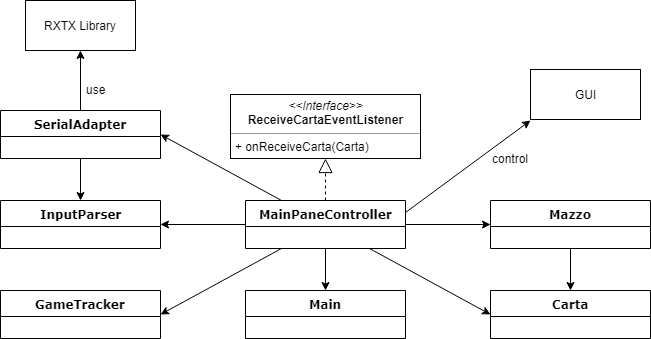
# Capitolo 5. Control Unit

# 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.

## Classi e relazioni tra le classi

Figura 8.



* *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, come chi detiene il turno, chi ha effettuato la presa della mano e genera il token in base allo stato del gioco.  
  Metodi principali:
  + public byte getToken();
* *MainPaneController:* motore della GUI, incorpora 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.

In questa applicazione, il Controller della GUI crea la classe InputParser e successivamente chiama il metodo *setEventListener* passando come argomento sé stesso (***this***). Quando 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.

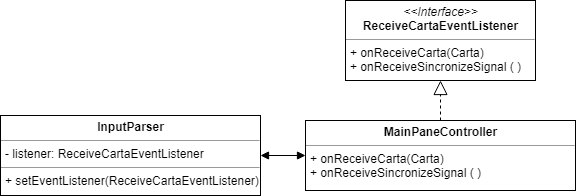
La presenza dell’interfaccia ReceiveCartaEventListener permette di rendere il codice riutilizzabile, cambiando solo l’implementazione dei metodi offerti dall’interfaccia.

Figura 9.

### 6.2.1 Implementazione del pattern Observer

Nel costruttore della classe MainPaneController viene creato 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 già 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 permette di disegnare tutte le interfacce grafiche presenti in questo progetto.  
Java nativamente non supporta l’interfaccia grafica nei suoi programmi, utilizza 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.   
SceneBuilder si basa su JavaFX, utilizza un file .xml (*Briscola.fxml*) in cui memorizza le informazioni relative alle interfacce grafiche e che viene caricato nel Main del programma.

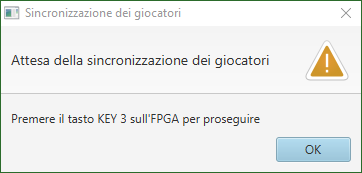
Il motore grafico MainPaneController utilizza dei tag nella dichiarazione degli oggetti grafici e nei metodi che scatenano eventi per permettere allo SceneBuilder di referenziarli e legare l’oggetto grafico al suo comportamento al determinarsi di un certo evento.

Tag: @FXML

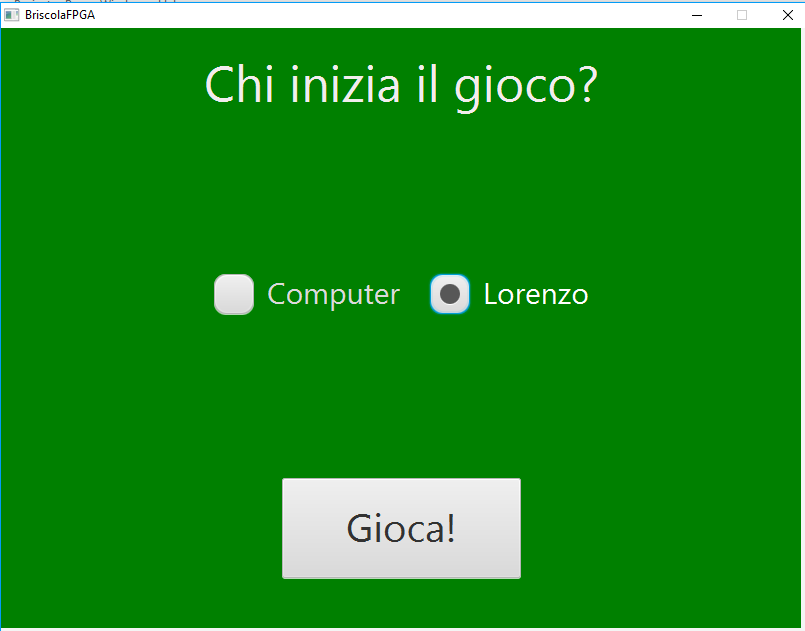
## 6.4 Funzionamento

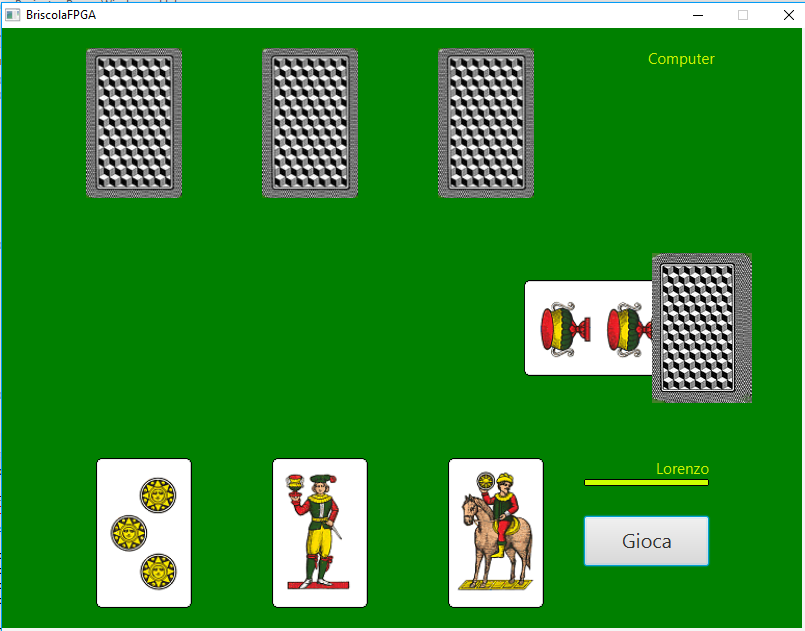
All’avvio della partita viene richiesto il nome del giocatore, che verrà poi visualizzato in fase di gioco.

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 comunicazione corretta.



Viene successivamente richiesto all’utente quale giocatore inizierà la partita.

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



# Capitolo 7. Punto d’incontro: comunicazione seriale

L’utente interagisce con la View, scritta in Java, la quale notifica alla DE1 tutti i cambiamenti effettuati. La comunicazione avviene tramite protocollo seriale UART.

## 7.1 UART

# Capitolo 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

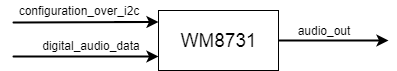
La DE1 contiene un codec stereo appositamente progettato per la lettura di file mp3, il WM8731.

Figura 10. 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

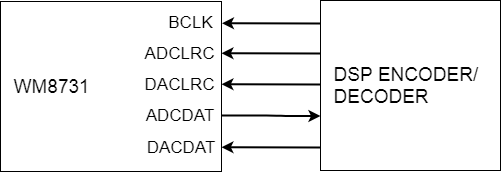
Il DSP (*Digital signal processor*), mostrato in figura 2, è un hardware dedicato e ottimizzato presente sulla scheda DE1 che permette di elaborare efficientemente segnali digitali.

Figura 11. WM8731 in SLAVE mode

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 I2C

L’I2C (*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).

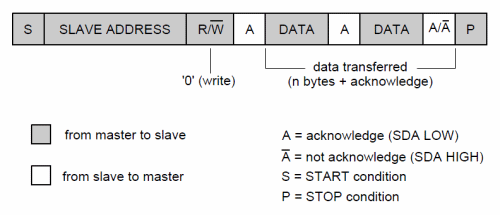
Di seguito è mostrato tramite l’uso di diagrammi temporali quanto appena spiegato.

Figura 4. Esempio di comunicazione I2C

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

Figura 3. Funzionamento del protocollo I2C

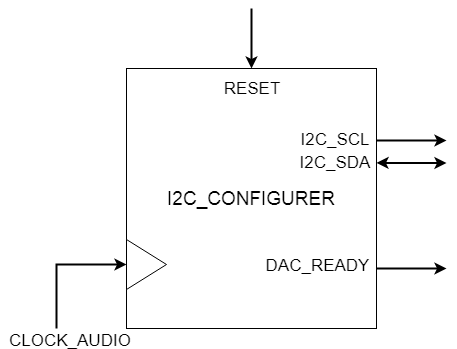
Il suo comportamento è stato definito utilizzando i seguenti *process*:

Figura 5. Entità I2C

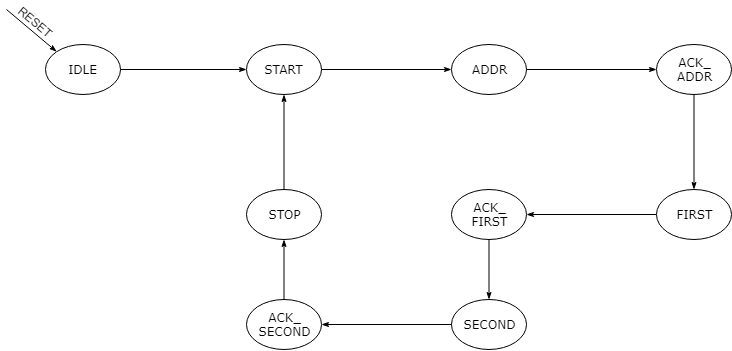
* I2CClockProcess : process(CLOCK\_12, RESET)
  + si occupa di generare, tramite l’utilizzo di un prescaler, i clock utili al protocollo e le transizioni tra i dati; DA METTERE CODICE
* FSMI2CProcess : process(CLOCK\_12, RESET)
  + definisce un automa a stati finiti che scandisce il regolare funzionamento del protocollo; a grandi linee, può essere visto nel seguente modo:

Figura 6. ASF dell’I2C\_codec

* 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

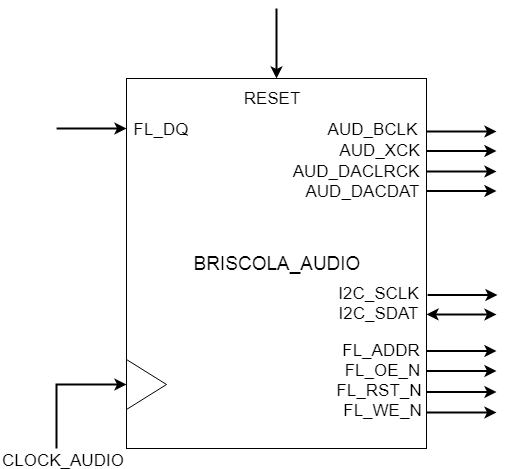
In base a quanto mostrato fino a questo punto, l’entità di top-level che si occuperà della configurazione del codec audio WM8731 (tramite l’utilizzo dell’I2C\_CONFIGURER) e della riproduzione dell’audio sarà la seguente:

Figura 7. 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
  + DA METTERE CODICE
* ReadAudioProcess : process(CLOCK\_AUDIO, RESET)

legge il file audio dalla memoria FLASH DA METTERE CODICE

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.

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.

Figura 8. System Control Panel DE1

# Capitolo 9. Fonti