# 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 Briscola5](#_Toc18877272)

[Capitolo 2. Architettura del sistema6](#_Toc18877273)

[Capitolo 3. Datapath7](#_Toc18877274)

[Capitolo 4. Control Unit20](#_Toc18877275)

[Capitolo 5. Graphic User Interface: il lato Java21](#_Toc18877276)

[Capitolo 6. Punto d’incontro: comunicazione seriale24](#_Toc18877277)

[Capitolo 7. Audio26](#_Toc18877278)

[7.1 Codec stereo WM873126](#_Toc18877279)

[7.2 Protocollo I2C27](#_Toc18877280)

[7.3 Interfaccia PCM31](#_Toc18877281)

[7.4 Entità di top-level32](#_Toc18877282)

[Capitolo 8. Fonti33](#_Toc18877283)

# Capitolo 1. La Briscola

## 1.1 ciao

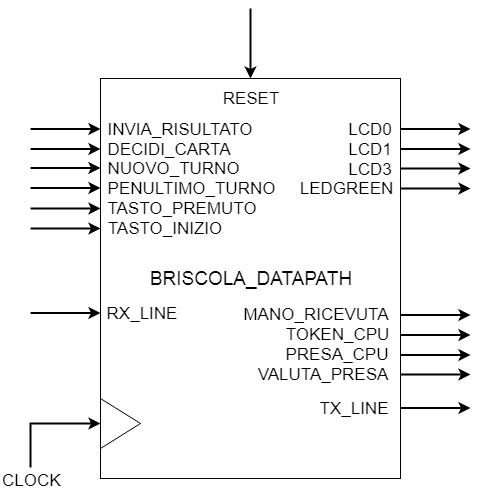
# Capitolo 2. Architettura del sistema

# Capitolo 3. 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

Schema generale del datapath:

  
Figura 1: Schema generale del 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.

## 3.1 Tipi di dato

All’interno del briscola\_package è presente la definizione dei tipi di dato utilizzati per definire le entità principali utilizzate. Di seguito la loro realizzazione in linguaggio VHDL:

-- Costanti

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

-- Tipi di dato

type vincitore is (GIOCATORE, CPU);   
-- definisce se il vincitore della partita sia o meno il giocatore piuttosto che l'FPGA

type seme is (BASTONI, DENARI, COPPE, SPADE);   
-- definisce il seme della carta

-- definisce il tipo carta con tutti i suoi attributi

type carta is record

numero : integer;

seme\_carta : seme;

valore : integer;

briscola : boolean;

end record;

type mazzo is array (0 to 39) of carta;   
-- definisce il numero di carte all'interno del mazzo

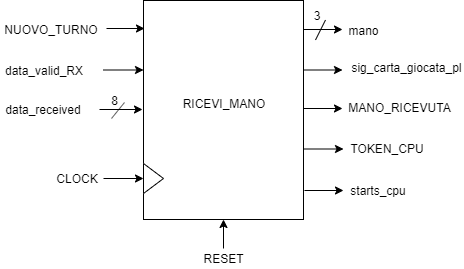
type mano\_cpu is array (0 to 2) of carta;   
-- definisce le tre carte che la CPU ha in mano in ogni momento

## 3.2 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.

Il componente riceve in ingresso i seguenti segnali:

  
Figura 2: Schema a blocchi del modulo di ricezione delle carte

* 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: registio che 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);

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;

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

sig\_carta\_giocata\_pl <= carta\_ricevuta; -- carta giocata dal player  
 stop\_ric\_mano := '0';  
 carta\_counter := 0;

elsif(carta\_in\_arrivo = '1') then

mano(indice\_carta\_giocata) <= carta\_ricevuta; -- carta da mettere nella mano  
 else  
 if(stop\_ric\_mano = '0') then

if(mano\_counter = 3) then

briscola\_partita <= carta\_ricevuta; -- briscola della partita  
 elsif(mano\_counter < 3) then

mano(mano\_counter) <= carta\_ricevuta; -- carta nella mano iniziale  
 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 =>

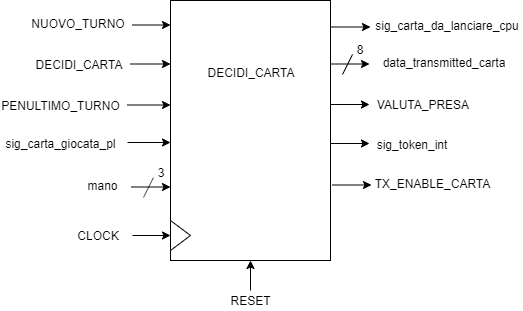
end case;

## 3.3 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:

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

  
Figura 3: Schema a blocchi del modulo di decisione della carta

* ***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;

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';

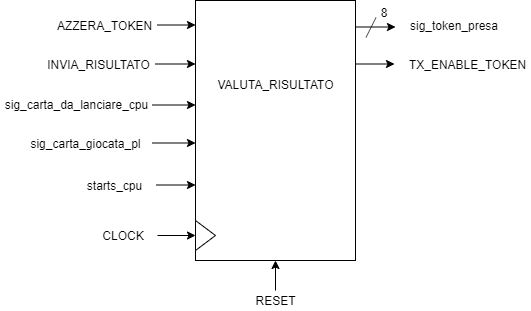
end if;

## 3.4 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:

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

  
Figura 4: Schema a blocchi del modulo di valutazione della presa

* ***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;

byte\_result := "00001111";   
 -- non tocca al giocatore, ha preso la CPU

else

punti\_player <= punti\_player +   
 sig\_carta\_da\_lanciare\_cpu.valore +  
 sig\_carta\_giocata\_pl.valore;

byte\_result := "01110000";   
 -- tocca al giocatore, ha preso il giocatore

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;

byte\_result := "01110000";   
 -- tocca al giocatore, ha preso il giocatore

else

punti\_cpu <= punti\_cpu + sig\_carta\_da\_lanciare\_cpu.valore + sig\_carta\_giocata\_pl.valore;

byte\_result := "00001111";   
 -- non tocca al giocatore, ha preso la CPU

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

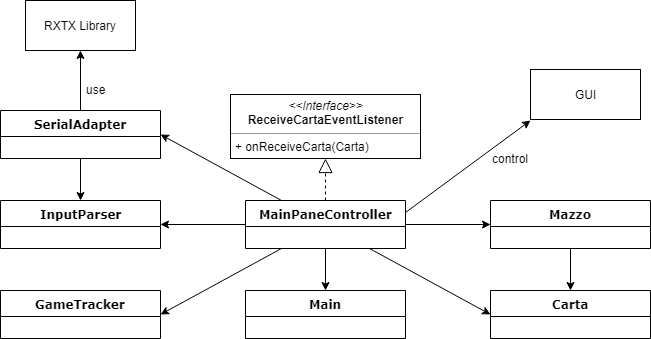
# Capitolo 4. Control Unit

# Capitolo 5. 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.

## 5.1 Classi e relazioni tra le classi

* 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()

  
Figura 5: Diagramma delle classi

**public** **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()

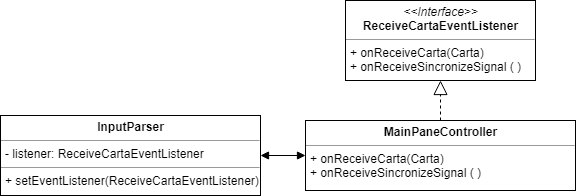
**public** Image getImageFromCarta(Carta c)

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

## 5.2 Pattern Observer

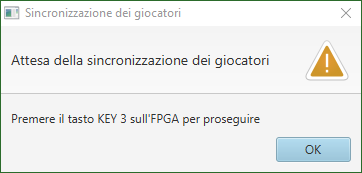
Il pattern Observer (Publish-Subscriber) viene utilizzato nei contesti in cui un oggetto richiede che altri oggetti aggiornino il proprio stato interno, notificandone il cambiamento.

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 6: Diagramma delle classi dell'applicazione del pattern Observer

## 5.3 Funzionamento

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

  
Figura 7: Alert di attesa della sincronizzazione

In questa fase preliminare è necessario far comunicare direttamente i due componenti per poter instaurare una comunicazione corretta.

# Capitolo 6. Punto d’incontro: comunicazione seriale

## 6.1 Libreria RXTX

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

### 6.1.1 Parametri di configurazione

**private** **static** **final** **int** ***DATA\_RATE*** = 9600;

serialPort.setSerialPortParams(***DATA\_RATE***,

SerialPort.***DATABITS\_8***,

SerialPort.***STOPBITS\_1***,

SerialPort.***PARITY\_NONE***);

### 6.1.2 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**);

### 6.1.3 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());

}

}

}

### 6.1.4 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]);

}

}

### 6.1.5 Scrittura dei dati sulla porta seriale

**public** **void** writeToSerialPort(**byte** out) {

System.***out***.print("\nInvio: ");

System.***out***.println(Integer.*toBinaryString*(out));

os.write(out);

}

# Capitolo 7. Audio

Per sfruttare al meglio le potenzialità della scheda, abbiamo deciso di inserire un sottofondo musicale che è possibile ascoltare durante la partita.

## 7.1 Codec stereo WM8731

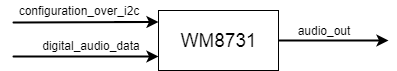
La DE1 contiene un codec stereo appositamente progettato per la lettura di file mp3, il WM8731.

Figura 8. 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*)

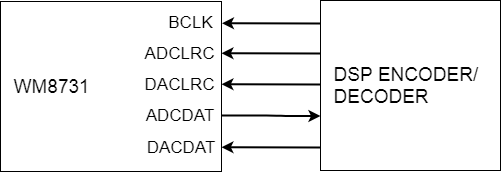
Il DSP (*Digital signal processor*), mostrato in figura 2, è un hardware dedicato e ottimizzato presente sulla scheda DE1 che permette di elaborare efficientemente segnali digitali.

Figura 9. 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*).

## 7.2 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).

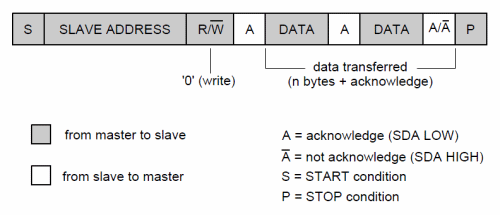
Di seguito è mostrato tramite l’uso di diagrammi temporali quanto appena spiegato.

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

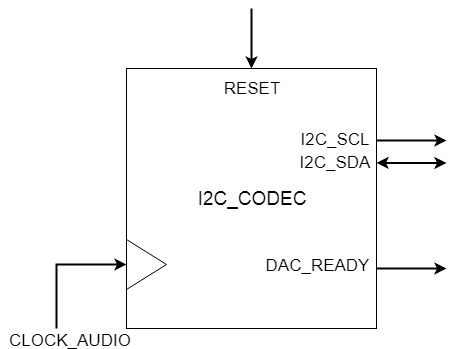
Il suo comportamento è stato definito utilizzando i seguenti *process*:

Figura 5. Entità I2C

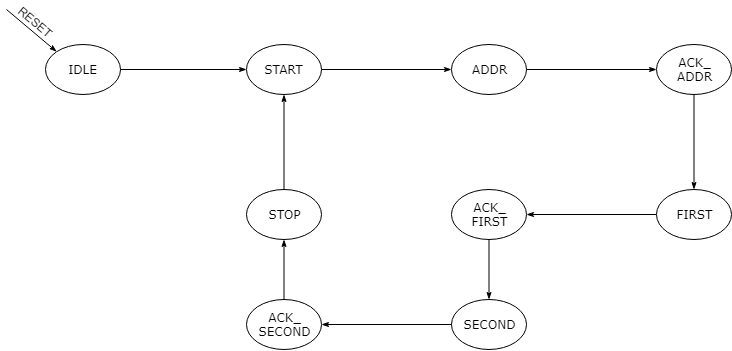
* I2CClockProcess : process(CLOCK\_12, RESET)
  + si occupa di generare, tramite l’utilizzo di un prescaler, i clock utili al protocollo e le transizioni tra i dati;
* FSMI2CProcess : process(CLOCK\_12, RESET)
  + definisce un automa a stati finiti che scandisce il regolare funzionamento del protocollo;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

Figura 6. ASF dell’I2C\_codec

* InitProcess : process(CLOCK\_12, RESET)
  + si occupa della configurazione vera e propria 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';

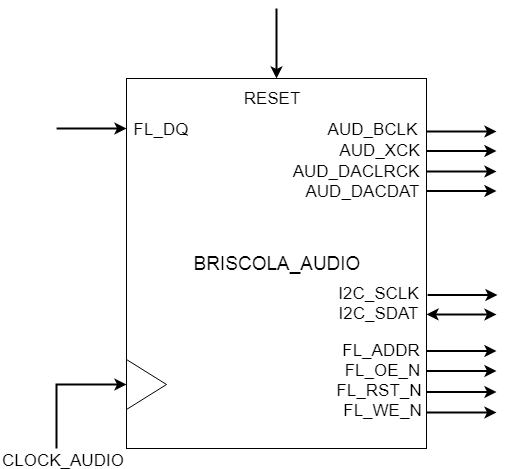
dac\_ready <= '1';

when others => … -- reset data and flag

end case;

## 7.3 Interfaccia PCM

## 7.4 Entità di top-level

In base a quanto mostrato fino a questo punto, l’entità di top-level che si occuperà della riproduzione dell’audio sarà la seguente:

# Capitolo 8. Fonti