

Progetto Perudo tramite FPGA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria Informatica

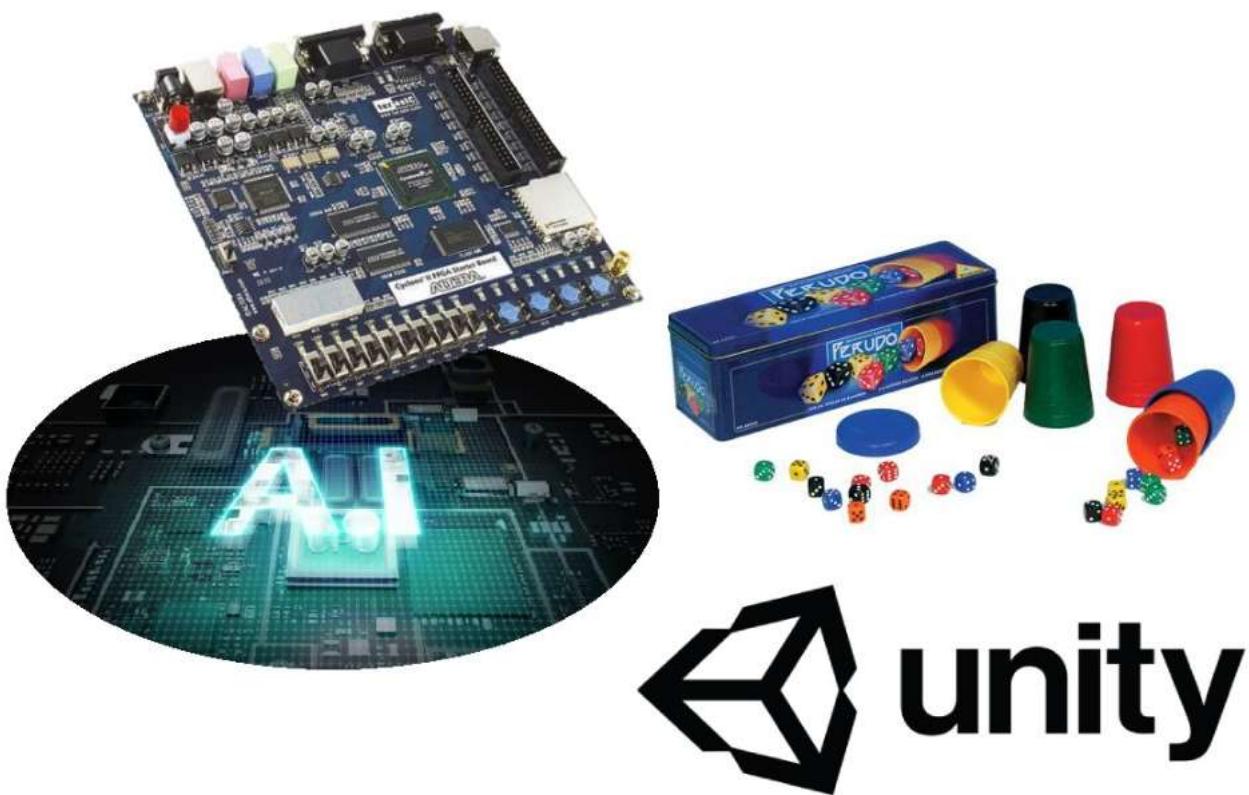
Progetto di Sistemi Digitali M – Prof. Eugenio Faldella

Caligiana Paolo

Bartoli Simone

Bonaccio Giovanni

Anno Accademico 2018/19



1 SOMMARIO

2	Introduzione.....	5
2.1	Introduzione a Perudo.....	5
2.2	Descrizione del problema	6
3	Strumenti utilizzati	7
4	Architettura.....	9
5	Perudo_Package	10
5.1	Valori Costanti.....	10
5.2	Tipi di Dato.....	10
5.3	Funzioni Ausiliarie	11
6	Perudo_Datapath	12
6.1	Generazione dado casuale	13
6.2	Aggiunta di un giocatore	16
6.3	Eliminazione di un giocatore	17
6.4	Rigenerazione dei dadi	18
6.5	Check.....	20
6.6	Gestione dei turni dei giocatori	21
6.7	Gestione scommesse.....	23
7	AI_Controller	25
7.1	Scelta della scommessa fpga	26
8	Perudo_Controller	31
8.1	Gestione dei pulsanti	32
8.2	Gestione degli stati	33
8.3	Interazione con Perudo_View_Controller.....	42
9	Perudo_View_Controller.....	43
9.1	MY_TX	43
9.2	Il protocollo di comunicazione.....	45
10	View	52
10.1	L'ambiente Unity.....	52

10.2	Perudo in Unity.....	53
11	Conclusioni.....	58
11.1	Risultati conseguiti	58
11.2	Problemi riscontrati	58
11.3	Sviluppi futuri e miglioramenti	59
12	Riferimenti.....	60

2 INTRODUZIONE

Lo scopo del progetto è simulare la versione del gioco da tavolo Perudo [1], utilizzando il linguaggio VHDL, realizzando un’architettura su FPGA [2] che permetta all’utente di giocare in tempo reale contro uno o più giocatori.

Ogni giocatore sarà comandato dall’FPGA e sarà possibile vedere, turno per turno, le scelte che l’FPGA stessa prende in base a determinate situazioni che si presentano durante la partita.

Come verrà spiegato successivamente, l’algoritmo con il quale l’FPGA prende le decisioni si basa su calcoli matematici probabilistici, i quali permetteranno ai giocatori di effettuare sempre la scelta migliore in base allo stato della partita.

2.1 INTRODUZIONE A PERUDO

Il Perudo è un gioco da tavolo per 2 o più giocatori in cui ogni giocatore inizia la partita con 5 dadi mescolati e coperti da un bussolotto.

Ogni giocatore guarda i propri dadi, tenendoli nascosti agli altri giocatori.

Nel Perudo i dadi sono considerati per il loro effettivo valore numerico su cinque facce (i numeri da 2 a 6) mentre sulla sesta faccia c’è il numero 1 che viene detto “Lama” e può valere anche come Jolly.

Se siete voi a cominciare, fate la vostra scommessa: scegliete un numero e dite quante volte, secondo voi, è uscito sui dadi tirati da tutti i giocatori; prima di fare la vostra scommessa, è necessario che teniate in mente due cose:

- Quanti dadi in totale sono in gioco.
- Tutte le Lama valgono come Jolly.

A questo punto il turno passa al giocatore alla vostra destra, il quale ha due possibilità:

- 1) Credere alla vostra scommessa e rilanciare.
- 2) Sfidarvi con il Dubito.

Rilancio

Se si decide di fare una scommessa più alta, ci sono tre possibilità:

- 1) La prima è scommettere che lo stesso numero scelto dal giocatore precedente compaia sui dadi per un totale di volte più alto rispetto a quanto scommesso da quel giocatore.
- 2) La seconda è scommettere che un numero superiore a quello su cui ha scommesso il giocatore precedente compaia sul totale dei dadi per un numero pari o superiore di volte.



Fig 1: Perudo

- 3) La terza è scommettere sui Lama, ovvero la vostra scommessa deve essere almeno pari a metà del numero di dadi scommessi dal giocatore precedente, se si tratta di un numero pari, oppure almeno la metà più uno, se si tratta di un numero dispari.

Dubito

Se non credete al giocatore precedente, potete dubitare della sua scommessa dicendo Dubito. Cominciando da chi sta alla vostra sinistra, ogni giocatore alza il proprio bussolotto e mostra i propri dadi. Si conta quante volte ricorre il numero in questione (compresi i Lama) fino a provare che avete ragione oppure torto. Se si scopre che il numero dubitato ricorre un numero di volte pari o maggiore di quanto scommesso dal giocatore precedente, perdete la scommessa e, di conseguenza, un dado. In caso contrario, siete voi a vincere la scommessa e sarà il giocatore precedente a perdere un dado. Alla fine della scommessa, tutti i giocatori mischiano nuovamente i propri dadi nei bussolotti. Sarà il giocatore che ha perso la scommessa a iniziare il nuovo turno.

Quando perdete il vostro ultimo dado uscite dal gioco e la partita continua con il giocatore alla vostra sinistra.

Lama

Se il vostro turno viene dopo quello di un giocatore che ha scommesso sui Lama, voi potete o scommettere su un numero più alto di Lama o tornare a scommettere sui numeri, nel qual caso la vostra scommessa deve essere almeno pari al doppio del numero di Lama su cui si è appena scommesso, più uno.

2.2 DESCRIZIONE DELLA SOLUZIONE

Dalla descrizione delle regole del gioco, si evince che l'implementazione in VHDL risulta non poco complessa. Perciò si è pensato di strutturare il problema in vari componenti, affidando ad ognuno il relativo sotto problema.

Si è deciso di suddividere i ruoli come segue:

- Gestore di tutti i dati della partita.
- Controllo centrale delle varie componenti.
- Gestore dell'intelligenza del giocatore FPGA.
- Interfaccia grafica.

Di rilevante importanza è la parte di gestione dell'intelligenza dell'FPGA, affidata ad un componente che implementa principi di Intelligenza Artificiale per la generazione della scommessa. Infine, l'interfaccia grafica è stata realizzata utilizzando il motore grafico Unity [3], implementando quindi un protocollo di comunicazione tra quest'ultimo e la Control Unit.

3 STRUMENTI UTILIZZATI

Per quanto riguarda lo sviluppo dei componenti del sistema, abbiamo suddiviso quest'ultimo in varie fasi, che insieme vanno a definire la seguente pipeline:

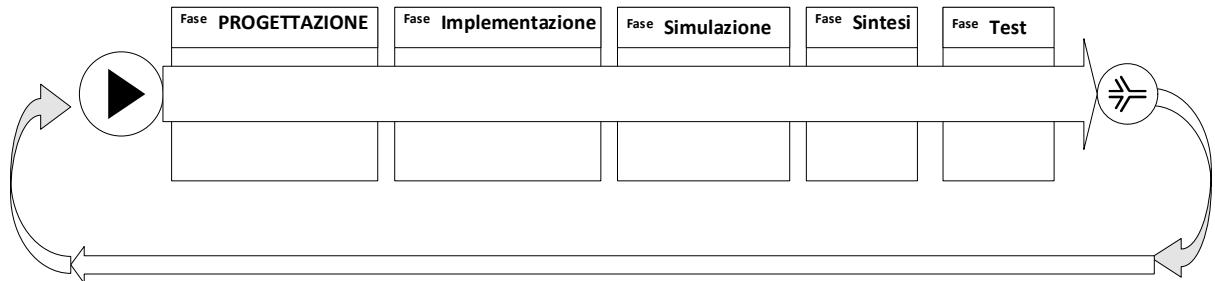


Fig 2: Pipeline di sviluppo

Per ognuna di queste fasi abbiamo utilizzato strumenti e ambienti di sviluppo diversi, illustrati come segue:

- **Altera Quartus II [4]**: per l'implementazione in VHDL
- **Unity**: per l'implementazione della View in C#
- **ModelSim [5]**: per la simulazione
- **Altera Quartus II**: per la fase di sintesi
- **Altera FPGA Cyclone II**: board sulla quale è stato sintetizzato la parte in VHDL
- **RS-232 [6]**: per la comunicazione tra FPGA e Unity

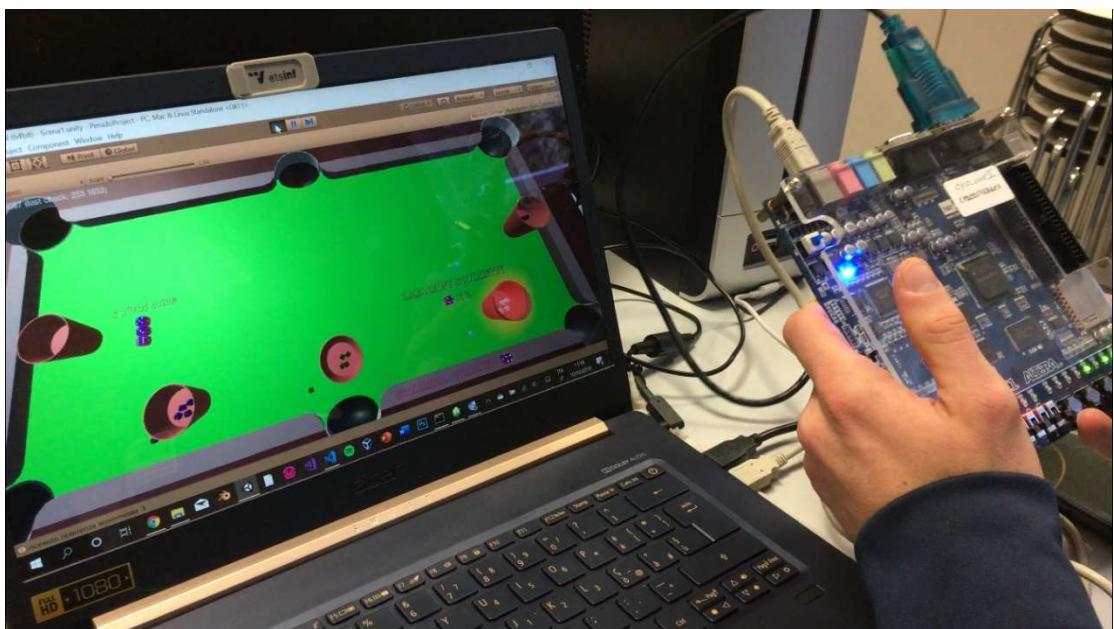


Fig 3: Foto esempio d'interfacciamento tra FPGA e Unity attraverso RS-232, a progetto ancora in fase di sviluppo. Codice sintetizzato sulla board attraverso Quartus.

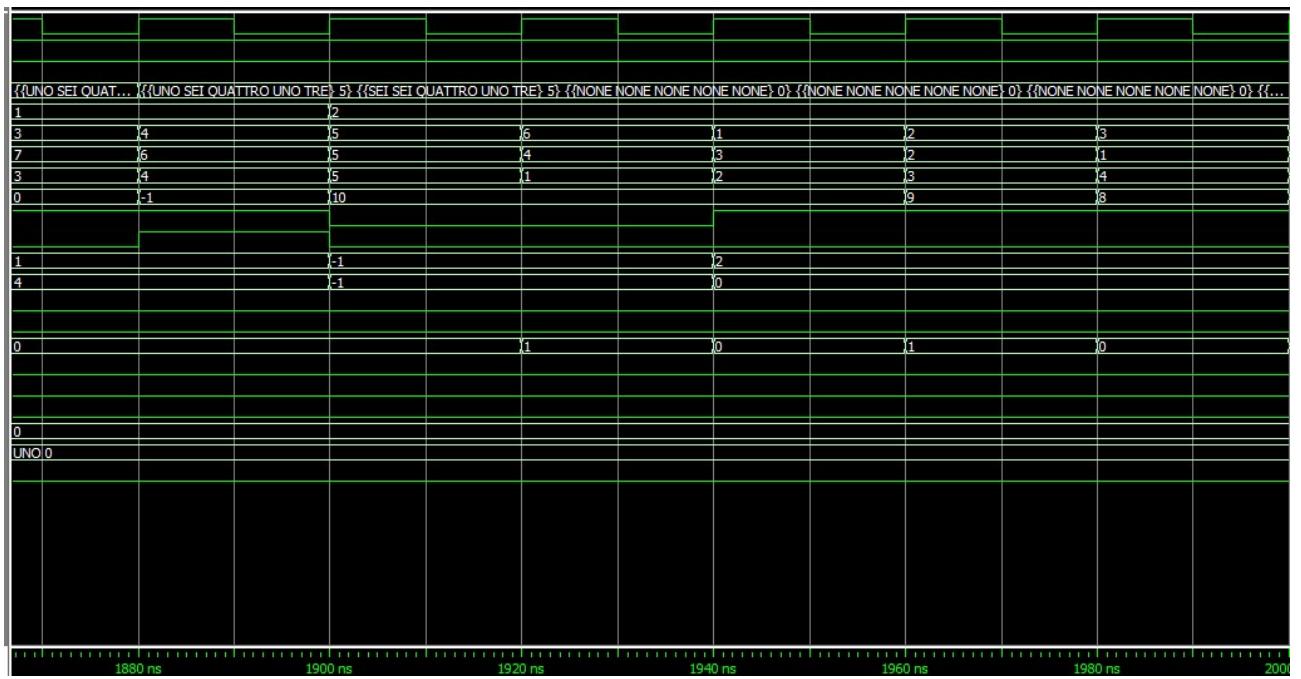


Fig 4: Screen di simulazione dell'operazione di AGGIUNTA GIOCATORE tramite ModelSim.

4 ARCHITETTURA

L'architettura generale si basa sul Pattern MVC:

- **Model:** corrisponde al nostro Perudo_Datapath, che gestisce l'insieme dei dati, le modifiche e risponde alle singole interrogazioni sui dati
- **View:** realizzata attraverso un'interfaccia Unity, riceve le informazioni da visualizzare mediante un canale seriale
- **Controller:** consiste nella Control Unit del sistema e corrisponde al nostro Perudo_Controller, gestisce le interazioni dei pulsanti e temporizza tutto il funzionamento del gioco, inviando/ricevendo segnali da/verso la View e il Model

Per resettare il sistema viene utilizzato lo switch SW9 presente sulla board FPGA, dopo di che sarà la Control Unit a sincronizzare tutte le varie reti logiche inviando gli opportuni segnali di inizializzazione.

I componenti e le reti logiche realizzate che andremo ad analizzare in questo report sono:

- Perudo_Package
- Perudo_Datapath
- AI_Controller
- Perudo_Controller
- Perudo_View_Controller
- MY_TX

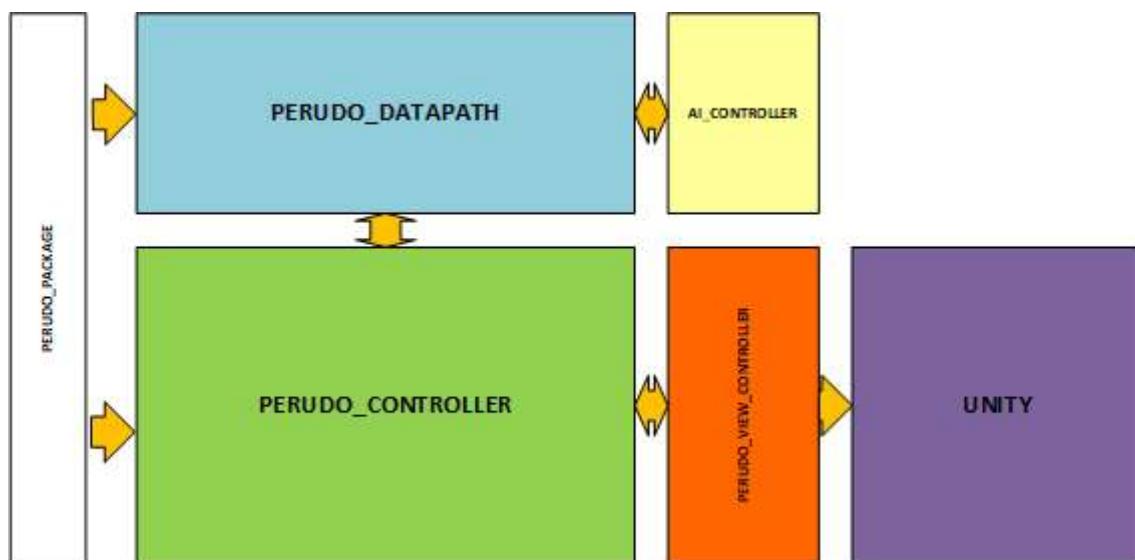


Fig 5: architettura del sistema

5 PERUDO_PACKAGE

Perudo_package è la parte in cui vengono definiti tutti i tipi di dato, i valori costanti e le funzioni necessarie all’implementazione della logica del gioco. Per questa ragione, come verrà illustrato in seguito, il contenuto di questo package è richiamato da quasi tutte le entità del sistema.

5.1 VALORI COSTANTI

- **MIN_DADI, MAX_DADI:** rappresentano il minimo e il massimo numero di dadi che può possedere un giocatore.
- **MIN_GIOCATORI, MAX_GIOCATORI:** rappresentano il minimo e massimo numero di giocatori previsti dal sistema.
- **VALORE_DADO_MIN, VALORE_DADO_MAX:** rappresentano il valore minimo e massimo che un dado può assumere.
- **MIN_ATTESA_CASUALE, MAX_ATTESA_CASUALE,**
MIN_ATTESA_CASUALE_2, MAX_ATTESA_CASUALE_2: costanti utili alla generazione del dado casuale, illustrata in seguito.
- **FATTORE_DUBITO, FATTORE_DUBITO_2P, FATTORE_DUBITO_3P,**
SOGLIA_DUBITO: costanti utili alla valutazione del fitness della scommessa corrente, illustrata in seguito.

5.2 TIPI DI DATO

- **dado_type:** definisce il tipo “dado” che può assumere i valori:
 - **UNO**
 - **DUE**
 - **TRE**
 - **QUATTRO**
 - **CINQUE**
 - **SEI**
 - **NONE**

Ognuno di essi rappresenta una facciata di un dado, mentre **NONE** indica un dado non inizializzato.

- **dado_array:** definisce un array di dado_type.
- **giocatore:** definisce il tipo giocatore formato da:
 - **dadi_in_mano:** dado_array di dimensione cinque. Rappresenta i cinque dadi associati al giocatore considerato.

- **numero_dadi_in_mano**: rappresenta il numero dei dadi associati al giocatore considerato.
- **giocatore_array**: definisce un array di giocatore.
- **scommessa_type**: definisce il tipo scommessa formato da:
 - **dado_scommesso**: dado_type, rappresenta il valore del dado scommesso.
 - **ricorrenza**: valore intero che rappresenta il numero di ricorrenze del dado considerato per la scommessa.
- **scommessa_array**: definisce un array di scommessa_type.

5.3 FUNZIONI AUSILIARIE

- **converti_da_intero_a_dado**: funzione utile agli altri componenti del sistema per convertire un intero in un dado_type.
- **converti_da_dado_a_intero**: funzione utile agli altri componenti del sistema per convertire un dado_type in un intero.
- **verifica_puntata**: funzione che verifica la validità di una scommessa nel rispetto delle regole del gioco.
- **isOddNumber**: funzione utile all'AI_Controller nel calcolo delle possibili mosse, in particolare restituisce “true” se un numero è dispari, “false” in caso contrario.

6 PERUDO_DATAPATH

Il Perudo_Datapath è la rete logica (sequenziale sincrona) che si occupa di tutta la gestione dei singoli dati, tenendo traccia di tutti i giocatori, i relativi dadi, del turno e della scommessa corrente. Esso infatti, si occupa dell'aggiunta ed eliminazione di un giocatore, della relativa generazione casuale dei dadi e della determinazione del turno all'inizio e dopo ogni puntata. Quest'ultime sono tutte funzionalità che vengono processate all'arrivo dei segnali in input da parte del Controller, come illustrato in seguito.

La comunicazione tra i vari sottocomponenti del Datapath avviene tramite un protocollo di Handshake, che consiste nella ricezione di un ACK di conferma di avvenuta operazione, successivo all'invio del segnale di input della relativa operazione.

Per la spiegazione del funzionamento di questa parte del sistema verranno illustrati solamente i segnali più significativi, al fine di una maggiore comprensione dei vari meccanismi.

Per ogni funzionalità del Datapath, verranno mostrati tutti i relativi componenti che cooperano; non verranno però considerati i componenti che restituiscono al controller i vari dati

(*I_TURNO_GIOCATORE, I_GIOCATORE_DADO_ELIMINATO, NR_GIOCATORI_IN_CAMPO, GIOCATORI*), in quanto funzionamento ritenuto alquanto intuitivo.



Fig 6: Macro-blocco del Datapath

```

entity Perudo_Datapath is
port
(
    CLOCK                                : in std_logic;
    RESET_N                               : in std_logic;

    -- Connections for the Controller

    -- Opzioni disponibili
    NUOVO_GIOCATORE                      : in std_logic;
    ELIMINA_GIOCATORE                     : in std_logic;

    PROSSIMO_TURNO                        : in std_logic;
    INIZIA_PARTITA                         : in std_logic;

    ESEGUI_SCOMMessa                      : in std_logic;
    DADO_SCOMMESsO                        : in dado_type;
    RICORRENZA                            : in integer;
    ESEGUI_SCOMMessa_FPGA                 : in std_logic;
    SCOMMessa_FPGA_DONE                   : out std_logic;
    SCOMMessa_DONE                         : out std_logic;

    RIGENERA                             : in std_logic;
    CHECK                                : in std_logic;

    RIGENERATI                           : out std_logic;
    CHECKED                              : out std_logic;
    GIOCATORE_AGGIUNTO                    : out std_logic;
    GIOCATORE_ELIMINATO                  : out std_logic;
    PARTITA_INIZIATA                      : out std_logic;
    FINE_PARTITA                          : out std_logic;
    I_GIOCATORE_DADO_ELIMINATO           : out integer := 0;

    DAMMI_TURNO_GIOCATORE                : in std_logic;
    I_TURNO_GIOCATORE                     : out integer range 0 to MAX_GIOCATORI-1;
    PROSSIMO_TURNO_ACK                   : out std_logic;

    DAMMI_GIOCATORI_IN_CAMPO             : in std_logic;
    NR_GIOCATORI_IN_CAMPO                : out integer range 0 to MAX_GIOCATORI;

    SCEGLI_SCOMMessa                     : out std_logic;
    SCOMMessa_ATTUALE                    : out scommessa_type;
    GIOCATORI                            : out giocatore_array(0 to MAX_GIOCATORI-1);
    SCOMMessa_SCELTA                     : in scommessa_type;
    SCOMMessa_OK                         : in std_logic;

    COUNTER                             : out integer := 0
);

end entity;

```

Fig 7: Definizione entità in VHDL del Perudo_Datapath

6.1 GENERAZIONE DADO CASUALE

La generazione del dado casuale è uno dei processi fondamentali del progetto. Ogni giocatore infatti, come mostrato nei paragrafi precedenti, è formato da una sequenza di dadi, i quali valori causano l'esito finale della partita intera. Per questo motivo è molto importante che essi vengano generati nella maniera più casuale possibile, al fine di rendere la partita più divertente e complessa da giocare.

Le componenti che realizzano il processo di generazione dei dadi sono il *GestoreDadi* e i tre contatori mostrati in Fig.8, il cui funzionamento sarà spiegato di seguito.

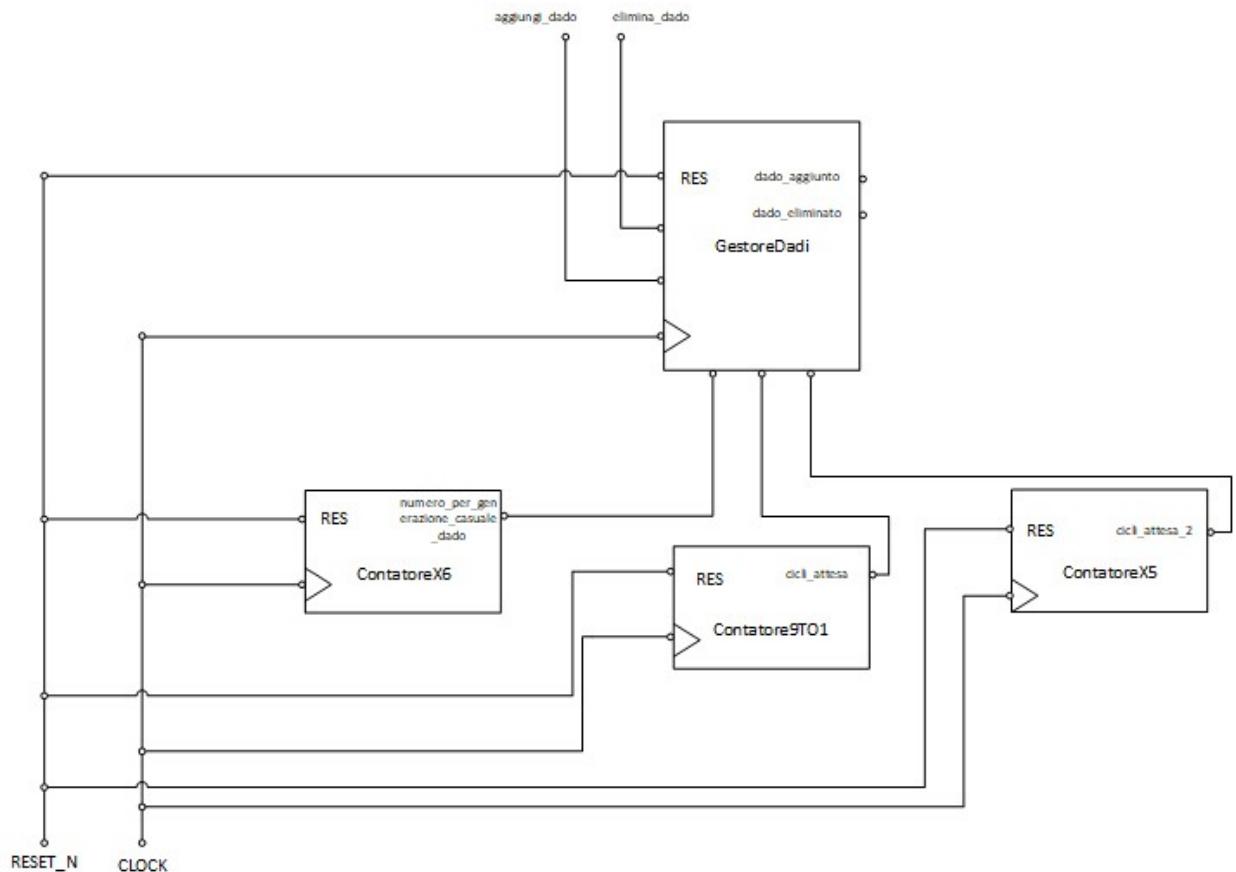


Fig 8: Schema circuitale dei componenti della GENERAZIONE DADO CASUALE

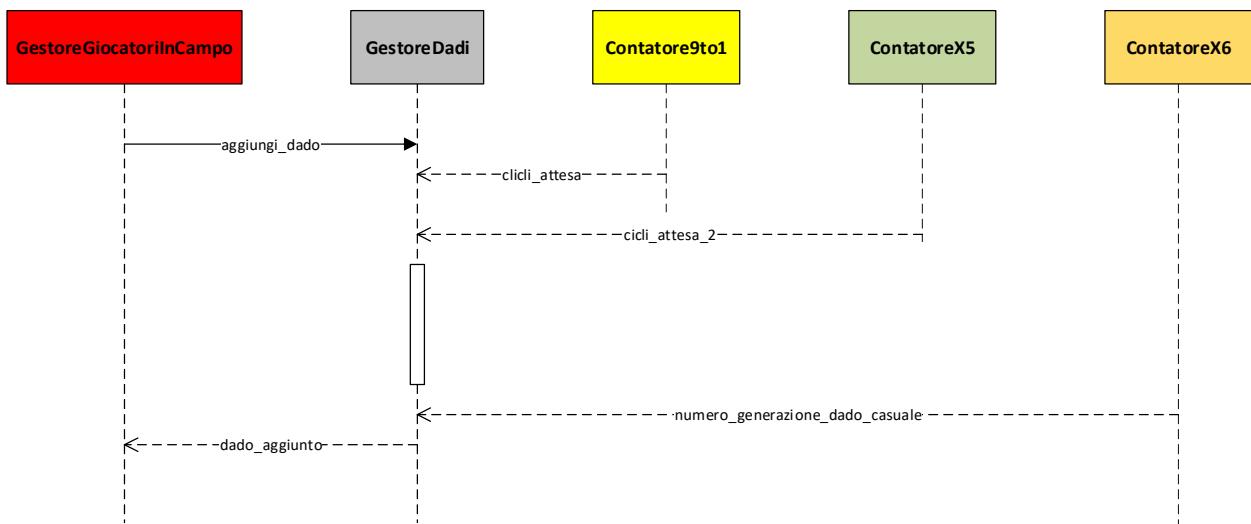


Fig 9: Diagramma delle interazioni nella GENERAZIONE DADO CASUALE

Il problema affrontato nell'implementazione di questo processo era dato dall'impossibilità di generare, per un determinato giocatore, cinque dadi aventi tutti valori diversi nello stesso ciclo di *CLOCK*. Per questo motivo si è deciso di temporizzare questa operazione. Il componente *GestoreDadi* infatti, alla ricezione del segnale *aggiungi_dado*, campiona il valore dei segnali in uscita dei contatori *Contatore9to1* e *ContatoreX5*, nominati rispettivamente *cicli_attesa* e *cicli_attesa_2*. Lo scopo è quello di attendere un numero di cicli di *CLOCK* pari alla somma dei valori dei due segnali, prima di campionare l'uscita dal *ContatoreX6*, che rappresenterà così il valore del dado. I dadi verranno aggiunti alla struttura dati *giocatori_in_campo*, di tipo *giocatore_array*, illustrato nei paragrafi precedenti. Infine, viene restituito il segnale di *ACK* chiamato *dado_aggiunto*.

Il ruolo del componente *ContatoreX6* è quello di contare in avanti di un'unità ad ogni ciclo di clock, per un intervallo compreso tra 1 e 6, rappresentando quindi il valore della facciata del dado.

Nel caso dei contatori *Contatore9to1* e *ContatoreX5* invece, il loro compito è quello di contare di un'unità ad ogni ciclo di clock. Il primo decrementa il segnale d'uscita per un intervallo compreso tra 9 e 1, mentre il secondo conta in avanti da 1 a 5. Come spiegato in precedenza, il valore del dado è campionato dopo *cicli_attesa* + *cicli_attesa_2* intervalli di clock, per rendere la distribuzione dei dadi il più casuale possibile. I valori 9 e 5 infatti, sono stati scelti in quanto, essendo due numeri dispari diversi, non permettono l'attesa di numeri di cicli di clock simmetrici o di valore ripetitivo. Per lo stesso motivo si è scelto di far contare uno dei due contatori all'indietro.

6.2 AGGIUNTA DI UN GIOCATORE

Questa operazione consiste nell'aggiunta di un giocatore al campo di gioco (fino ad un massimo di 8) e quindi nella generazione dei relativi cinque dadi, come illustrato nei paragrafi precedenti.

Le componenti che realizzano il processo di creazione di un giocatore sono *GestoreDadi* e *GestoreGiocatoriInCampo* mostrati in Fig.10, il cui funzionamento sarà spiegato di seguito.

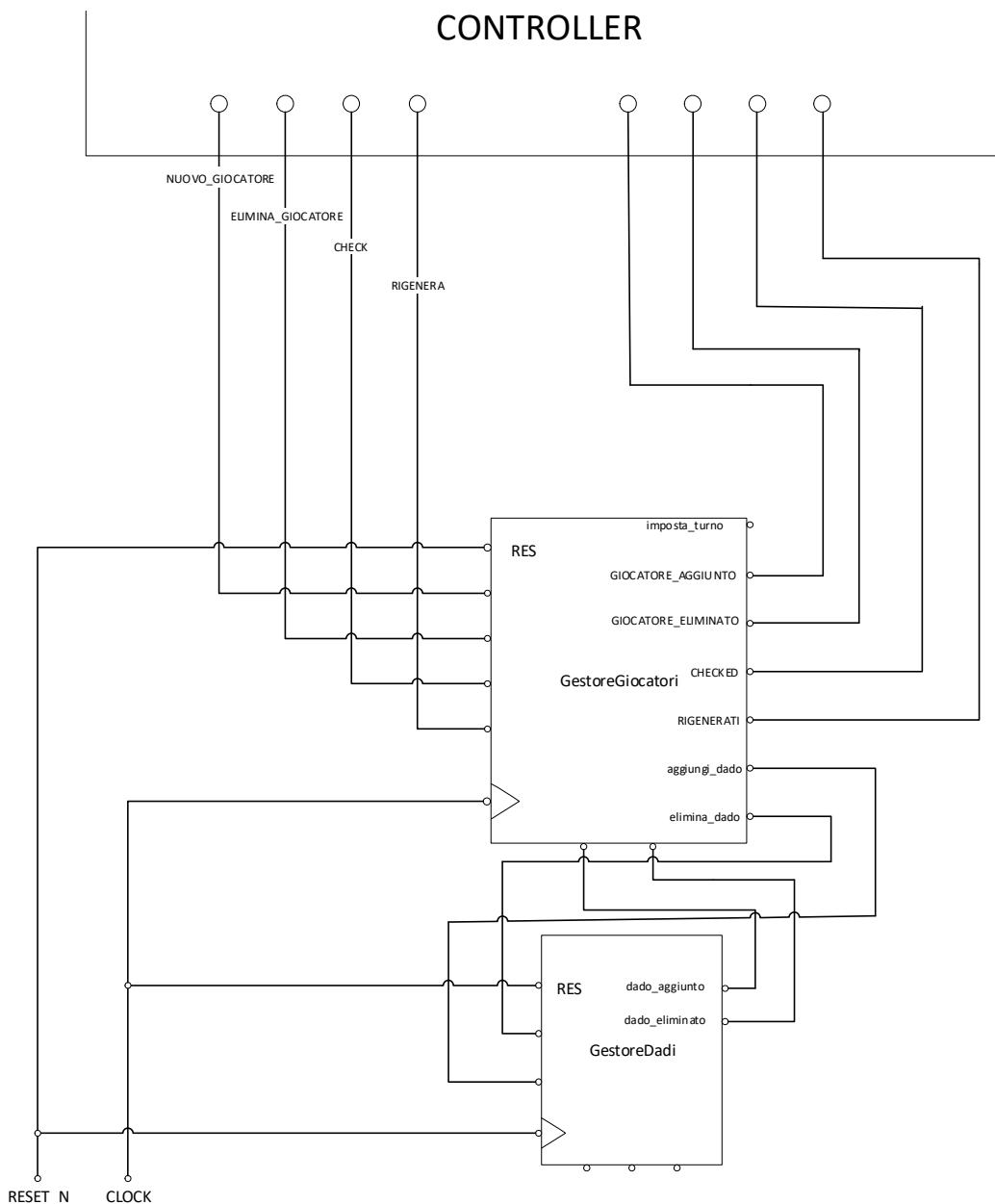


Fig 10: Schema circuitale dei componenti delle operazioni AGGIUNTA GIOCATORE, ELIMINAZIONE GIOCATORE, CHECK,

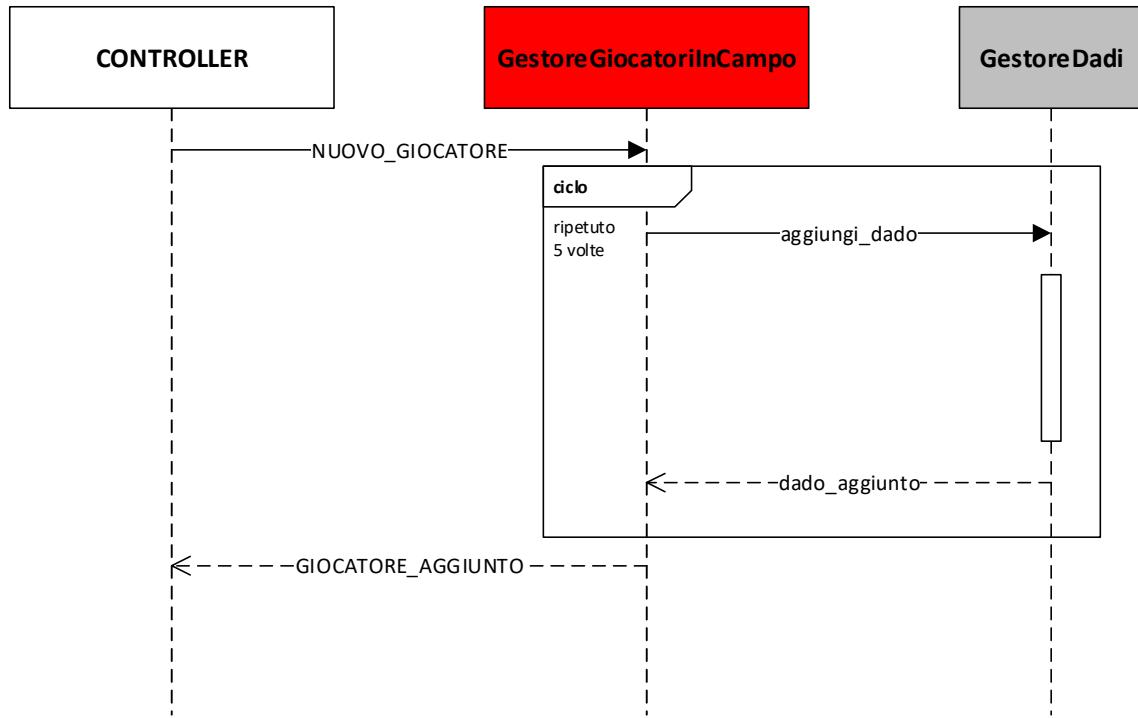


Fig 11: Diagramma delle interazioni nella operazione di AGGIUNTA GIOCATORE

Alla ricezione del segnale `NUOVO_GIOCATORE`, il componente `GestoreGiocatoriInCampo` richiama attraverso il segnale `aggiungi_dado` il componente `GestoreDadi`, che procederà all'aggiunta di un dado come già mostrato. Questo passo viene ripetuto cinque volte, in quanto un giocatore a inizio partita possiede un numero di dadi pari a cinque. Alla fine di questa iterazione, il componente `GestoreGiocatoriInCampo` restituirà un ACK al controller chiamato `GIOCATORE_AGGIUNTO`.

6.3 ELIMINAZIONE DI UN GIOCATORE

Questa operazione consiste nell'eliminazione di un giocatore dal campo di gioco e quindi nella rimozione dei relativi cinque dadi.

Le componenti che realizzano il processo di eliminazione del giocatore sono `GestoreDadi` e `GestoreGiocatoriInCampo` mostrati in Fig.5, il cui funzionamento sarà spiegato di seguito.

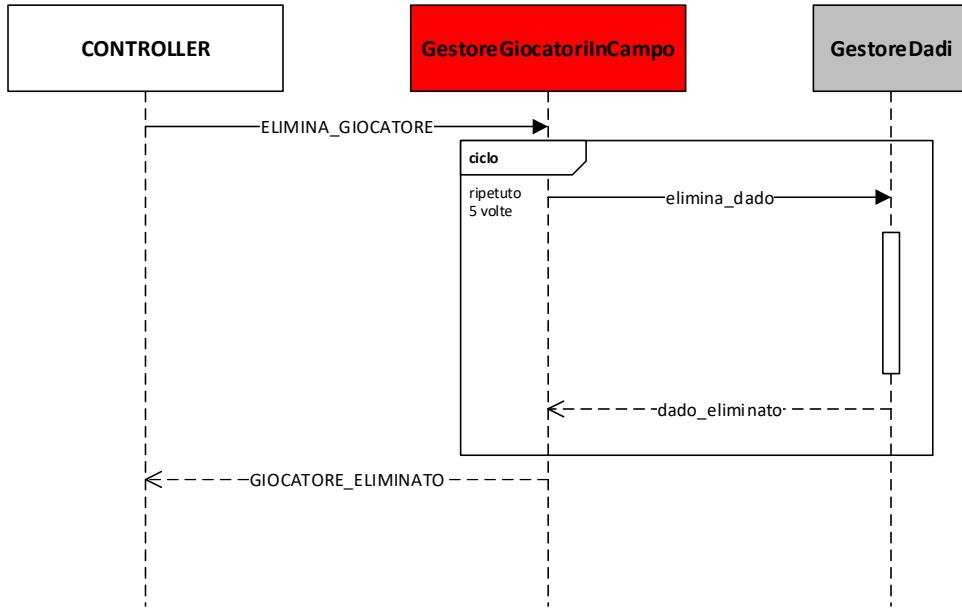


Fig 12: Diagramma delle interazioni nella operazione di ELIMINAZIONE GIOCATORE

Alla ricezione del segnale *ELIMINA_GIOCATORE*, il componente *GestoreGiocatoriInCampo* richiama attraverso il segnale *elimina_dado* il componente *GestoreDadi*, che procederà all’eliminazione dell’ultimo dado dell’ultimo giocatore (gestione LIFO) aggiunto alla struttura dati *giocatori_in_campo*. Dopo aver effettuato l’eliminazione del dado, il componente *GestoreDadi* restituirà un ACK chiamato *dado_eliminato*. Questo passo viene ripetuto cinque volte, in quanto un giocatore a inizio partita possiede un numero di dadi pari a cinque. Alla fine di questa iterazione, il componente *GestoreGiocatoriInCampo* restituirà un ACK al controller chiamato *GIOCATORE_ELIMINATO*.

6.4 RIGENERAZIONE DEI DADI

Questa operazione consiste nella rigenerazione dei dadi all’inizio di ogni round, considerando solamente i giocatori rimasti in campo fino a quel momento e i dadi rimasti a ognuno di essi. Anche in questo caso, l’obbiettivo è quello di distribuire in maniera più casuale possibile i valori dei dadi, soprattutto rispetto al round precedente. Per lo sviluppo di questo procedimento infatti, è stato scelto di richiamare il processo illustrato nel paragrafo 6.1, in quanto risultato adatto anche in questo caso.

Le componenti che realizzano il processo di rigenerazione dei dadi sono *GestoreDadi* e *GestoreGiocatoriInCampo* mostrati in Fig.13, il cui funzionamento sarà spiegato di seguito.

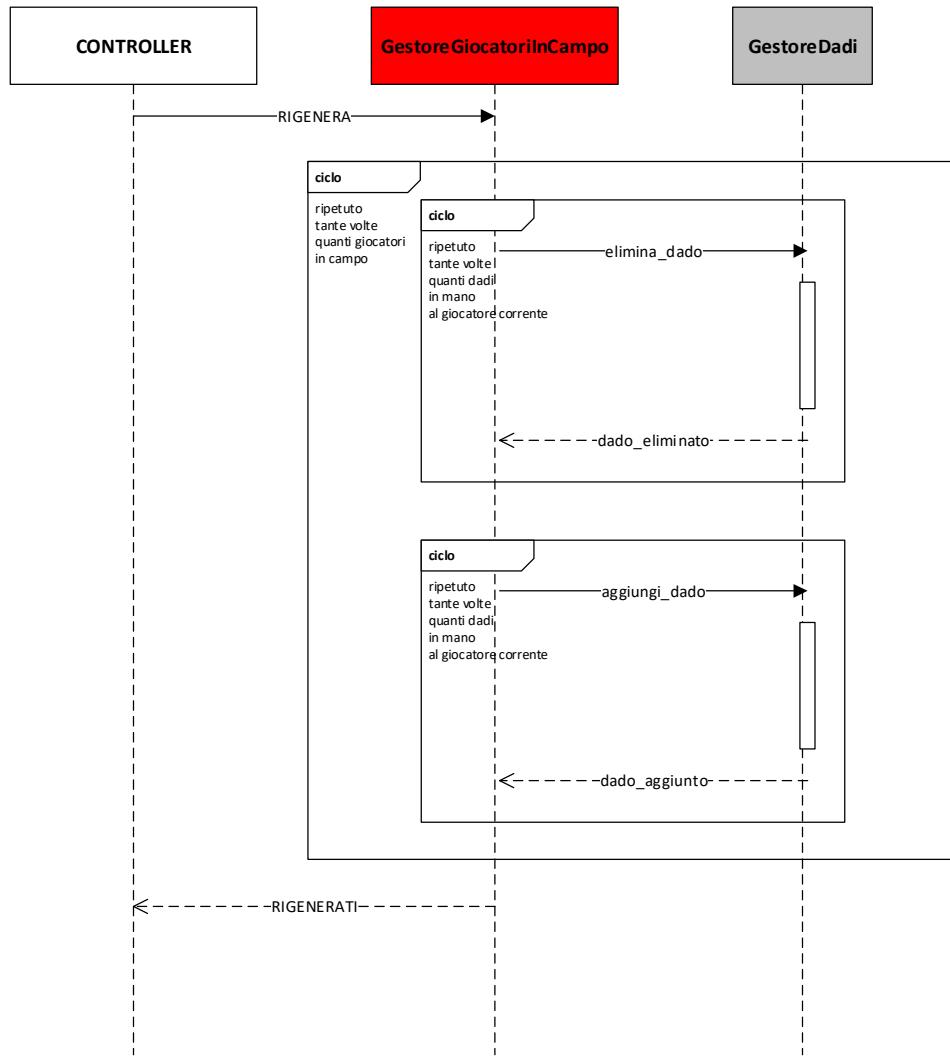


Fig 13: Diagramma delle interazioni nella operazione di RIGENERAZIONE DEI DADI

Alla ricezione del segnale *RIGENERA*, il componente *GestoreGiocatoriInCampo* richiama attraverso il segnale *elimina_dado* il componente *GestoreDadi*, che procederà all'eliminazione del dado del giocatore attraverso gli indici relativi. Questo passo viene ripetuto tante volte, quanti sono i dadi posseduti dal giocatore corrente. Alla fine di questa iterazione, il componente *GestoreGiocatoriInCampo* richiama attraverso il segnale *aggiungi_dado* il componente *GestoreDadi*, che procederà all'aggiunta di un dado al giocatore passato attraverso gli indici relativi. Questo passo viene ripetuto tante volte, quanto il doppio dei dadi posseduti dal giocatore corrente, in quanto i dadi vengono sia eliminati che aggiunti nuovamente. Questo processo viene iterato per tutti i giocatori rimasti in campo. Alla fine del ciclo, il componente *GestoreGiocatoriInCampo* restituirà un ACK al controller chiamato *RIGENERATI*.

6.5 CHECK

Questa operazione ha inizio quando uno dei giocatori scommette DUBITO e consiste nella verifica dell'ultima scommessa e nell'eliminazione di un dado al giocatore perdente. Inoltre, questo è il processo che può causare la fine della partita, come verrà illustrato in seguito.

Le componenti che realizzano il processo di check sono *GestoreDadi* e *GestoreGiocatoriInCampo* mostrati in Fig.14, il cui funzionamento sarà spiegato di seguito.

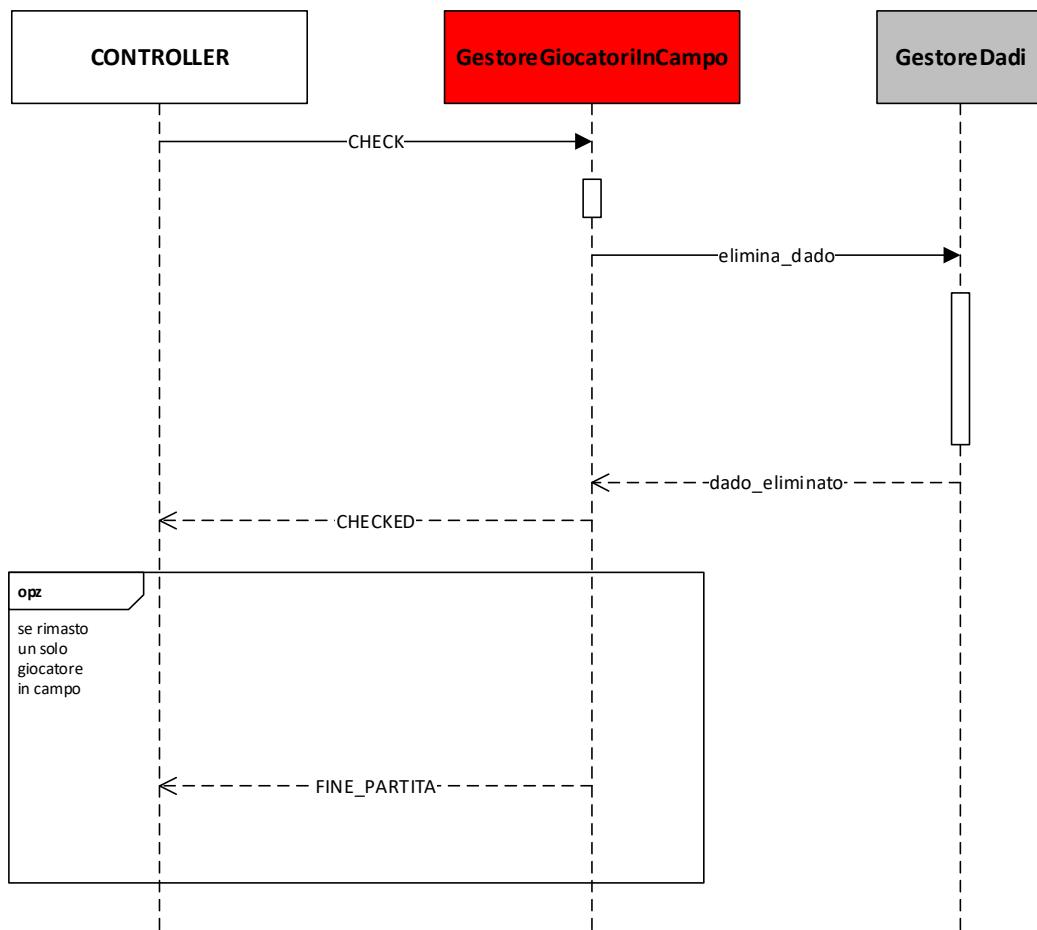


Fig 14: Diagramma delle interazioni nella operazione di CHECK

Alla ricezione del segnale **CHECK**, il componente *GestoreGiocatoriInCampo* verifica la validità dell'ultima scommessa effettuata, procedendo quindi col conteggio delle ricorrenze del dado scommesso. Se quest'ultime dovessero essere di un valore maggiore o uguale alla ricorrenza scommessa, il giocatore che effettua il DUBITO perde un dado, altrimenti è perso da chi ha scommesso per ultimo. Dopo aver verificato chi è stato sconfitto, il componente *GestoreGiocatoriInCampo* richiama *GestoreDadi* attraverso il segnale **elimina_dado**. Dopo l'esecuzione dell'eliminazione, *GestoreGiocatoriInCampo* restituirà un ACK al controller chiamato **CHECKED**. Verrà infine reimpostato il turno dei giocatori, come vedremo poi in

seguito. A questo punto, solo nel caso in cui dovesse esser rimasto un solo giocatore in campo, il componente *GestoreGiocatoriInCampo* invierà un segnale di fine partita al controller chiamato *FINE_PARTITA*.

6.6 GESTIONE DEI TURNI DEI GIOCATORI

Questo processo si occupa della gestione del turno dei giocatori durante tutto il corso della partita, quindi per tutti i round.

La componente che realizza questo processo è rappresentata dal *GestoreTurnoPartita* mostrato in Fig.15, il cui funzionamento sarà spiegato di seguito.

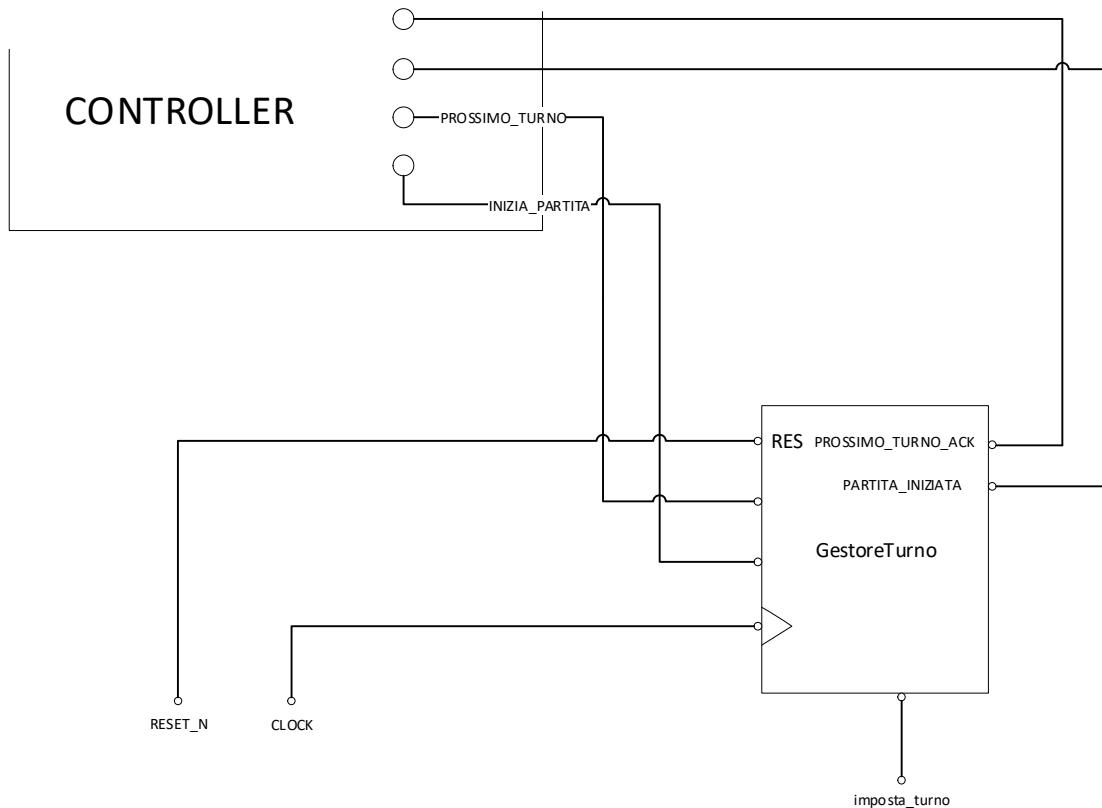


Fig 15: Schema circuitale dei componenti dell'operazione di GESTIONE TURNO

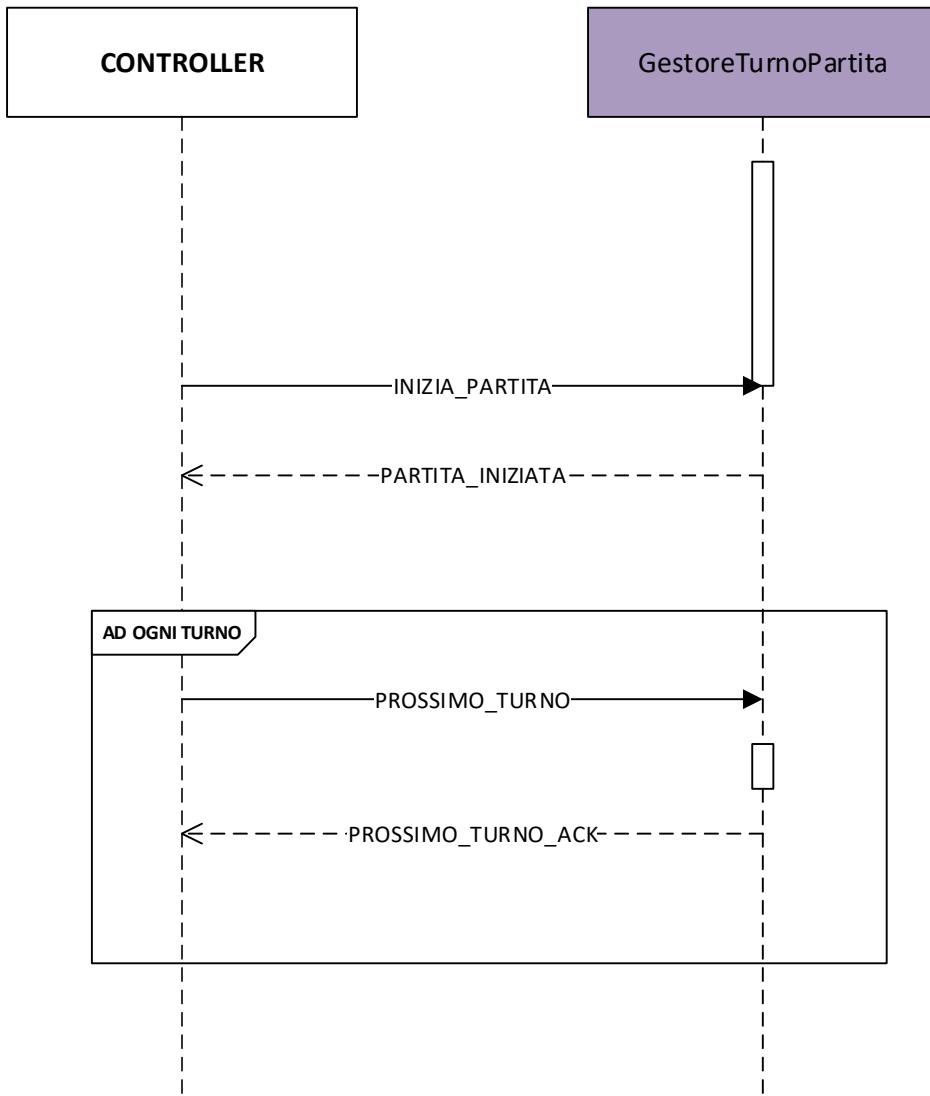


Fig 16: Diagramma delle interazioni nella operazione di GESTIONE TURNO

Dopo il reset del sistema, il componente *GestoreTurnoPartita* incrementa di un'unità l'indice del turno del giocatore ad ogni ciclo di clock. Alla ricezione del segnale `INIZIA_PARTITA`, il conteggio si blocca e viene restituito un ACK al controller chiamato `PARTITA_INIZIATA`. Da questo punto in poi il conteggio andrà in avanti di un'unità solo dopo la ricezione del segnale `PROSSIMO_TURNO`, in cui verrà restituito anche in questo caso un ACK al controller chiamato `PROSSIMO_TURNO_ACK`. Alla fine di ogni operazione di `CHECK` inoltre, il componente *GestoreTurnoPartita* riceverà il segnale `imposta_turno` dal *GestoreGiocatoriInCamp*, che gli permetterà di settare il turno all'ultimo giocatore perdente.

6.7 GESTIONE SCOMMESSE

Questa operazione consiste nel gestire la scommessa dell'Utente e la scommessa generata automaticamente dal sistema. Il componente realizzato per l'implementazione di questo processo è chiamato *GestoreScommesse* (Fig.17), che si occupa di memorizzare la scommessa dell'Utente ricevuta dal controller e di andare a ricavare dall'AI_Controller, la scommessa generata automaticamente, quando il turno è dell'FPGA. Nel primo caso semplicemente, alla ricezione del segnale *ESEGUI_SCOMMESSA*, il componente non fa altro che campionare i segnali *RICORRENZA* e *DADO_SCOMMESSO* ricevuti dal controller, e salvarli quindi nella struttura dati *scommessa_corrente* di tipo *scommessa_type*. Il secondo caso invece, verrà illustrato nei paragrafi successivi.

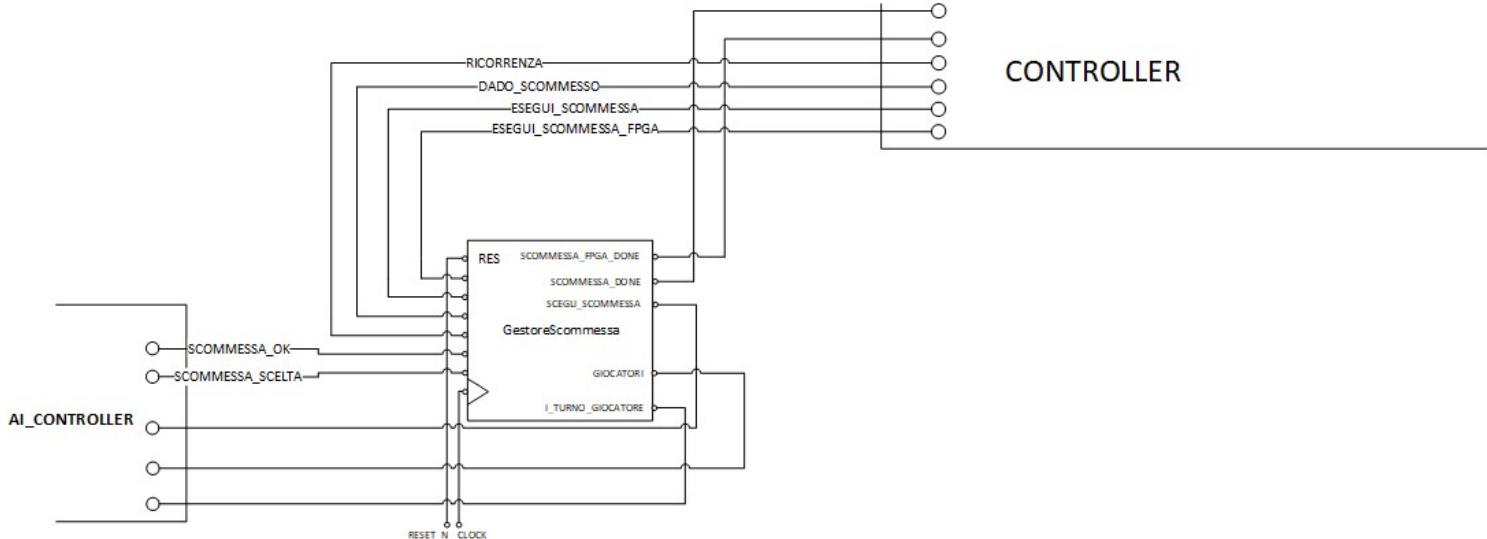


Fig 17: Schema circuitale del componente *GestoreScommesse*

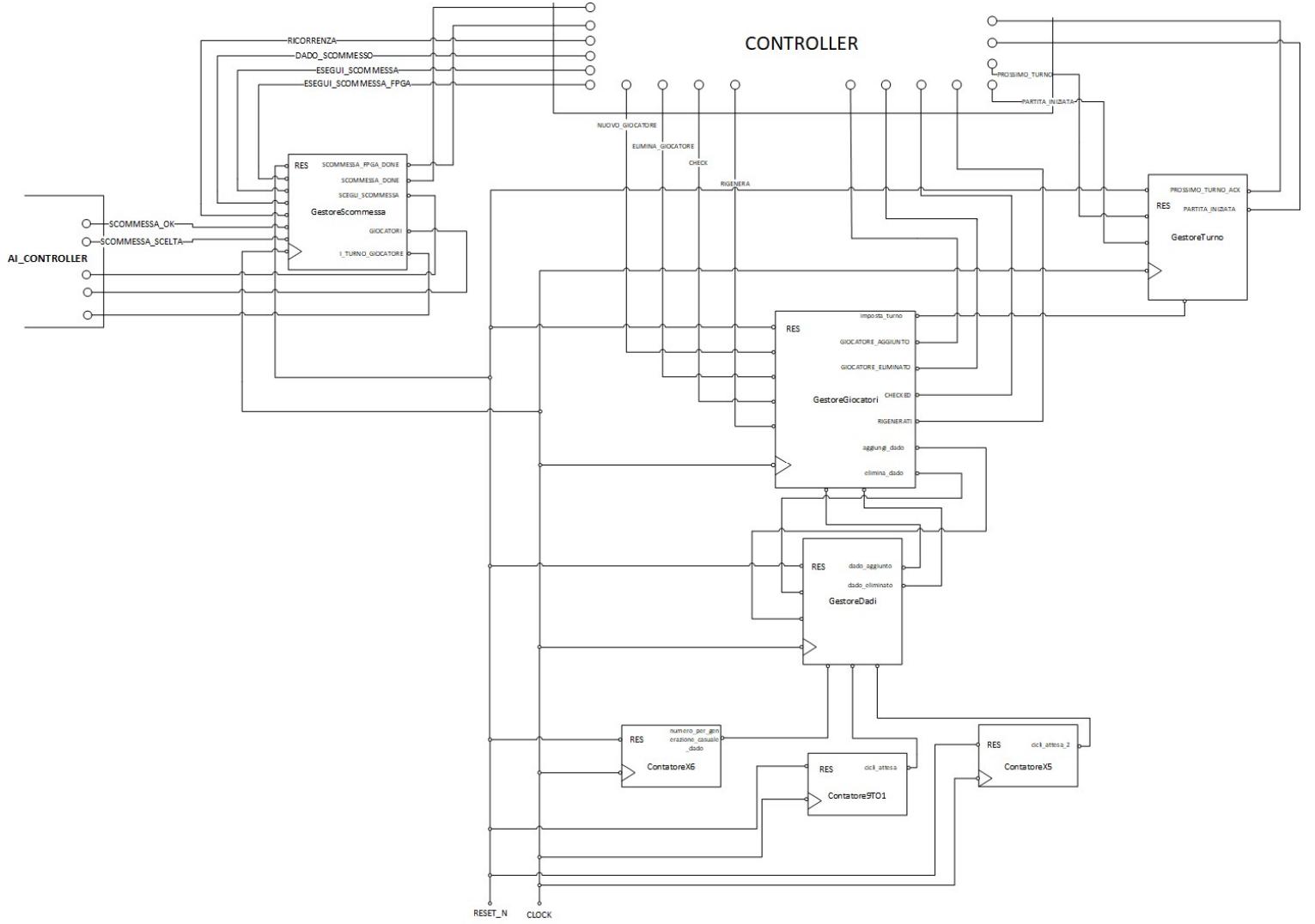


Fig 18: Schema circuitale Perudo_Datapath

7 AI_CONTROLLER

L'AI_Controller è la rete logica (sequenziale sincrona) che si occupa della generazione automatica della scommessa quando è il turno dell'FPGA. A tal fine, si è deciso di realizzare una rete dedicata in quanto abbiamo pensato di applicare principi di Intelligenza Artificiale per la scelta della migliore scommessa.

Questa parte del sistema è formata da un solo componente chiamato *ScegliScommessa* e da tre reti combinatorie ausiliarie, come mostrato in Fig.19.

Per la spiegazione del funzionamento di questa parte del sistema verranno illustrati solo i segnali più significativi al fine di una maggiore comprensione dei vari meccanismi.

L'interfacciamento col Perudo_Datapath avviene attraverso un protocollo di Handshake, che consiste nella ricezione di un ACK di conferma di avvenuta operazione, successivo all'invio del segnale di input dell'operazione relativa.



Fig 19: Macro-blocco dell'AI_Controller

```
-- -----
Entity AI_Controller is
-- -----
port
(
    CLOCK                      : in std_logic;
    RESET_N                     : in std_logic;

    -- Connection with Datapath
    SCEGLI_SCOMMESA             : in std_logic;
    SCOMMESSA_ATTUALE           : in scommessa_type;
    GIOCATORI                   : in giocatore_array(0 to MAX_GIOCATORI-1);
    I_TURNO_GIOCATORE          : in integer;

    SCOMMESSA_OK                : out std_logic;
    SCOMMESSA_SCELTA            : out scommessa_type
);

End Entity;
```

Fig 20: Definizione entità in VHDL dell'AI_Controller

7.1 SCELTA DELLA SCOMMESSA FPGA

Questa operazione consiste nel generare la scommessa automatica quando il turno corrisponde all'indice di uno dei giocatori dell'FPGA. Per applicare i principi di AI, si è pensato di delegare i compiti di generazione delle possibili mosse, calcolo fitness per ogni scommessa e calcolo fitness della scommessa precedente a tre reti combinatorie diverse, come verrà illustrato in seguito.

Le componenti che implementano questo processo sono rappresentate dal *ScegliScommessa* e dalle reti combinatorie interne al componente chiamate *dammi_tutte_le_azioni_posibili_scommesse*, *calcola_fitness* e *calcola_fitness_dubito*, come mostrato in Fig. 21.

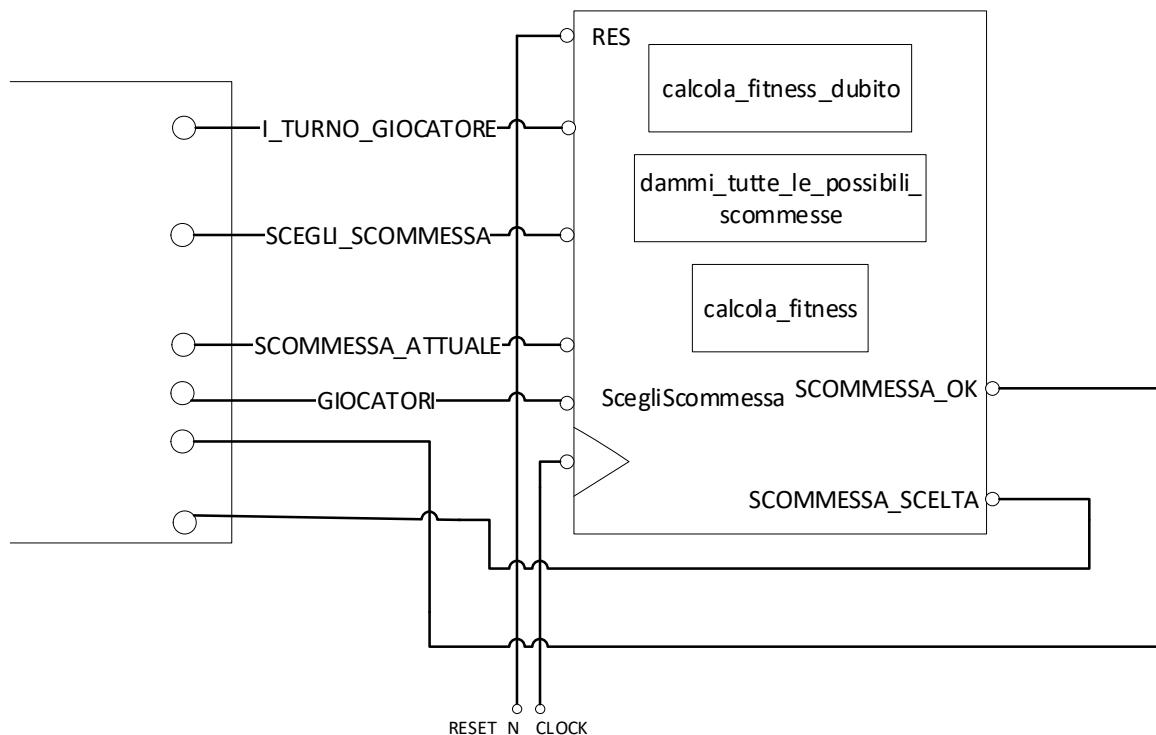


Fig 21: Schema circuitale AI_Controller

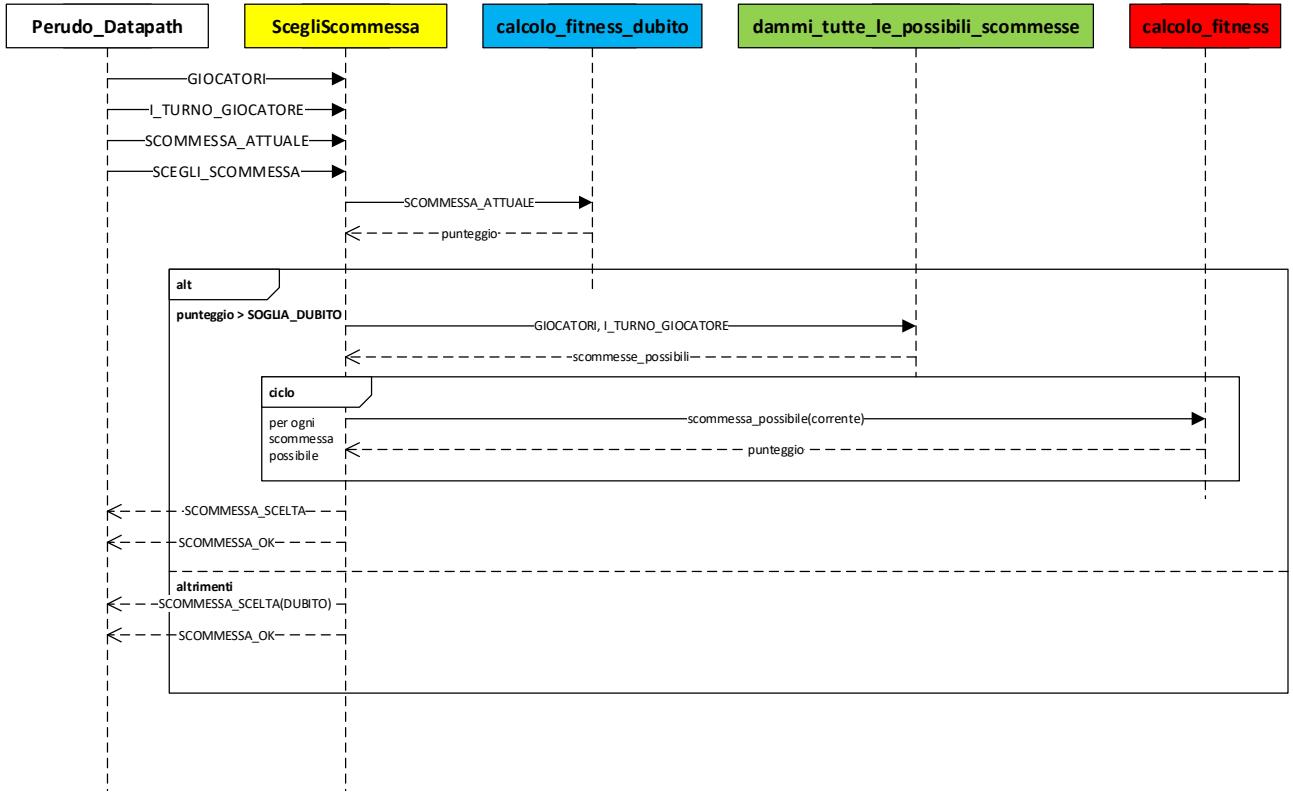


Fig 22: Diagramma delle interazioni nella operazione di SCELTA SCOMMESSA

Alla ricezione del segnale *SCEGLI_SCOMMESA*, il componente *ScegliScommessa* richiamerà la sotto rete *calcolo_fitness_dubito* con l’obiettivo di valutare il fitness della scommessa corrente, attraverso una funzione d’euristica dedicata e implementata nella sotto rete. Nel caso in cui il punteggio non dovesse superare il valore *SOGLIA_DUBITO* (definito nel Perudo_Package), il componente restituirà la scommessa DUBITO inviando inoltre un segnale di ACK al Datapath chiamato *SCOMMESSA_OK*. In caso contrario invece, verrà richiamata la sotto rete *dammi_tutte_le_azioni_posibili* che genererà alcune delle scommesse più plausibili nel rispetto delle regole, considerando la scommessa attuale e i dadi posseduti dal giocatore corrente. Delle scommesse restituite ne verrà valutato il fitness attraverso una funzione d’euristica dedicata e implementata nella sotto rete *calcola_fitness*. Verrà quindi selezionata la scommessa col punteggio più alto che verrà restituita al Datapath insieme al segnale di ACK *SCOMMESSA_OK*.

```

ScegliScommessa_RTL : process(CLOCK,RESET_N, SCEGLI_SCOMMESSA) is
variable dadi_totali_in_campo : integer;
variable dado_tmp : dado_type;
variable ricorrenza_tmp : integer;
variable punteggio_tmp : integer;
variable punteggio_max : integer;
variable scommesse_posibili : scommessa_array(VALORE_DADO_MIN to VALORE_DADO_MAX);
variable punteggio_dubito : integer;
begin
  if(RESET_N = '0') then
    SCOMMESSA_SCELTA.dado_scommesso <= NONE;
    SCOMMESSA_SCELTA.ricorrenza <= -1;
    dado_tmp := NONE;
    ricorrenza_tmp := -1;
    punteggio_tmp := 0;
    punteggio_max := -1;
    dadi_totali_in_campo := 0;
    SCOMMESSA_OK <= '0';
    punteggio_dubito := 0;
  elsif(rising_edge(CLOCK)) then
    SCOMMESSA_OK <= '0';
    if(SCEGLI_SCOMMESSA = '1') then
      if(dadi_totali_in_campo = 0) then
        dadi_totali_in_campo := dadi_in_campo(GIOCATORI);
      end if;
      punteggio_dubito := calcola_fitness_dubito(GIOCATORI(I_TURNO_GIOCATORE), SCOMMESSA_ATTUALE, dadi_totali_in_campo);
      if(punteggio_dubito > SOGLIA_DUBITO or SCOMMESSA_ATTUALE.ricorrenza = 0) then
        scommesse_posibili := dammi_tutte_le_azioni_posibili(SCOMMESSA_ATTUALE, dadi_totali_in_campo);
        for i in VALORE_DADO_MIN to VALORE_DADO_MAX loop
          if(scommesse_posibili(i).dado_scommesso /= NONE) then
            punteggio_tmp := calcola_fitness(GIOCATORI(I_TURNO_GIOCATORE), scommesse_posibili(i));
            if(punteggio_tmp>punteggio_max) then
              punteggio_max := punteggio_tmp;
              dado_tmp := scommesse_posibili(i).dado_scommesso;
              ricorrenza_tmp := scommesse_posibili(i).ricorrenza;
            end if;
          end if;
        end loop;
        SCOMMESSA_SCELTA.dado_scommesso <= dado_tmp;
        SCOMMESSA_SCELTA.ricorrenza <= ricorrenza_tmp;
        SCOMMESSA_OK <= '1';
        dadi_totali_in_campo := 0;
        dado_tmp := NONE;
        ricorrenza_tmp := -1;
        punteggio_tmp := 0;
        punteggio_max := -1;
        punteggio_dubito := 0;
      else
        SCOMMESSA_SCELTA.dado_scommesso <= NONE;
        SCOMMESSA_SCELTA.ricorrenza <= -1;
        SCOMMESSA_OK <= '1';
        dadi_totali_in_campo := 0;
        dado_tmp := NONE;
        ricorrenza_tmp := -1;
        punteggio_tmp := 0;
        punteggio_max := -1;
        punteggio_dubito := 0;
      end if;
    end if;
  end if;
end process;

```

Fig 23: Implementazione VHDL del componente scegli_scommessa

```

function dammi_tutte_le_azioni_posibili(SCOMMESSA_ATTUALE : scommessa_type; dadi_totali_in_campo : integer)
return scommessa_array is
variable dado_scommesso           : integer |;
variable totali_scommesse          : scommessa_array(VALORE_DADO_MIN to VALORE_DADO_MAX);
variable totali_scommesse_dim      : integer := 1;
begin
  for i in VALORE_DADO_MIN to VALORE_DADO_MAX loop
    totali_scommesse(i).ricorrenza := 0;
    totali_scommesse(i).dado_scommesso := NONE;
  end loop;
  dado_scommesso := converti_da_dado_a_intero(SCOMMESSA_ATTUALE.dado_scommesso);
  if(dado_scommesso /= 1) then
    for i in VALORE_DADO_MIN+1 to VALORE_DADO_MAX loop
      if(i >= dado_scommesso) then
        totali_scommesse(totali_scommesse_dim).dado_scommesso := converti_da_intero_a_dado(i);
        if(i = dado_scommesso) then
          if((SCOMMESSA_ATTUALE.ricorrenza)/=dadi_totali_in_campo) then
            totali_scommesse(totali_scommesse_dim).ricorrenza := SCOMMESSA_ATTUALE.ricorrenza + 1;
          else
            totali_scommesse(totali_scommesse_dim).dado_scommesso := NONE;
          end if;
        elsif(i /= VALORE_DADO_MAX ) then
          totali_scommesse(totali_scommesse_dim).ricorrenza := SCOMMESSA_ATTUALE.ricorrenza;
        else
          totali_scommesse(totali_scommesse_dim).dado_scommesso := NONE;
        end if;
      end if;
      totali_scommesse_dim := totali_scommesse_dim + 1;
    end loop;
    -- scommessa lama
    totali_scommesse(totali_scommesse_dim).dado_scommesso := UNO;
    if(isOddNumber(SCOMMESSA_ATTUALE.ricorrenza)) then
      totali_scommesse(totali_scommesse_dim).ricorrenza := (SCOMMESSA_ATTUALE.ricorrenza/2)+1;
    else
      totali_scommesse(totali_scommesse_dim).ricorrenza := (SCOMMESSA_ATTUALE.ricorrenza/2);
    end if;
    totali_scommesse_dim := totali_scommesse_dim + 1;
  else
    for i in VALORE_DADO_MIN+1 to VALORE_DADO_MAX loop
      totali_scommesse(totali_scommesse_dim).dado_scommesso := converti_da_intero_a_dado(i);
      totali_scommesse(totali_scommesse_dim).ricorrenza := (SCOMMESSA_ATTUALE.ricorrenza * 2)+1;
      totali_scommesse_dim := totali_scommesse_dim + 1;
    end loop;
    -- scommessa lama
    totali_scommesse(totali_scommesse_dim).dado_scommesso := UNO;
    totali_scommesse(totali_scommesse_dim).ricorrenza := SCOMMESSA_ATTUALE.ricorrenza + 1;
    totali_scommesse_dim := totali_scommesse_dim + 1;
  end if;
  return totali_scommesse;
end function;

```

Fig 24: Implementazione VHDL della rete combinatoria *dammi_tutte_azioni_posibili*

```

function calcola_fitness(giocatore : giocatore; scommessa : scommessa_type) return integer is
variable punteggio      : integer := 0;
variable ricorrenza     : integer := 0;
begin
    ricorrenza := ricorrenza_dado(scommessa.dado_scommesso, giocatore.dadi_in_mano);
    if(ricorrenza>=scommessa.ricorrenza) then
        punteggio := punteggio + 90;
    elsif(ricorrenza<scommessa.ricorrenza
          and ricorrenza>0) then
        punteggio := punteggio + (10*ricorrenza);
    else
        if(scommessa.dado_scommesso = UNO) then
            punteggio := punteggio + 50;
        end if;
    end if;
    return punteggio;
end function;

```

Fig 25: Implementazione VHDL della rete combinatoria *calcola_fitness*

```

function calcola_fitness_dubito(giocatore : giocatore; scommessa : scommessa_type; dadi_totali_in_campo : integer) return integer is
variable punteggio      : integer := 0;
variable dadi_ignoti     : integer;
variable ricorrenza_ignota : integer;
begin
    dadi_ignoti:= dadi_totali_in_campo - giocatore.numero_dadi_in_mano;
    ricorrenza_ignota:=scommessa.ricorrenza - ricorrenza_dado(scommessa.dado_scommesso, giocatore.dadi_in_mano);
    case dadi_totali_in_campo is
        when 2 to 7 => punteggio := (dadi_ignoti - ricorrenza_ignota)*FATTORE_DUBITO_2P;
        when 8 to 15 => punteggio := (dadi_ignoti - ricorrenza_ignota)*FATTORE_DUBITO_3P;
        when others => punteggio := (dadi_ignoti - ricorrenza_ignota)*FATTORE_DUBITO;
    end case;
    return punteggio;
end function;

```

Fig 26: Implementazione VHDL della rete combinatoria *calcola_fitness_dubito*

8 PERUDO_CONTROLLER

Il Perudo_Controller consiste nella Control Unit del sistema e si occupa di sincronizzare tutti i segnali e tutti gli eventi. Per far ciò, utilizza una gestione a stati gerarchica, dove, in base allo stato attuale del sistema, si aspetta di ricevere determinati segnali e consente o meno la pressione di determinati pulsanti.

Con gestione degli stati in maniera gerarchica si intende la presenza di un insieme di stati principali che a loro volta prevedono tanti sottostati secondari. Gli stati principali sono chiamati *internal_state* e si occupano di gestire tutte le fasi di gioco di una partita, mentre i vari sottostati secondari sono chiamati: *initialization_state*, *turn_player_state*, *turn_fpga_state* e *check_state* e consentiranno di temporizzare le varie sottofasi.

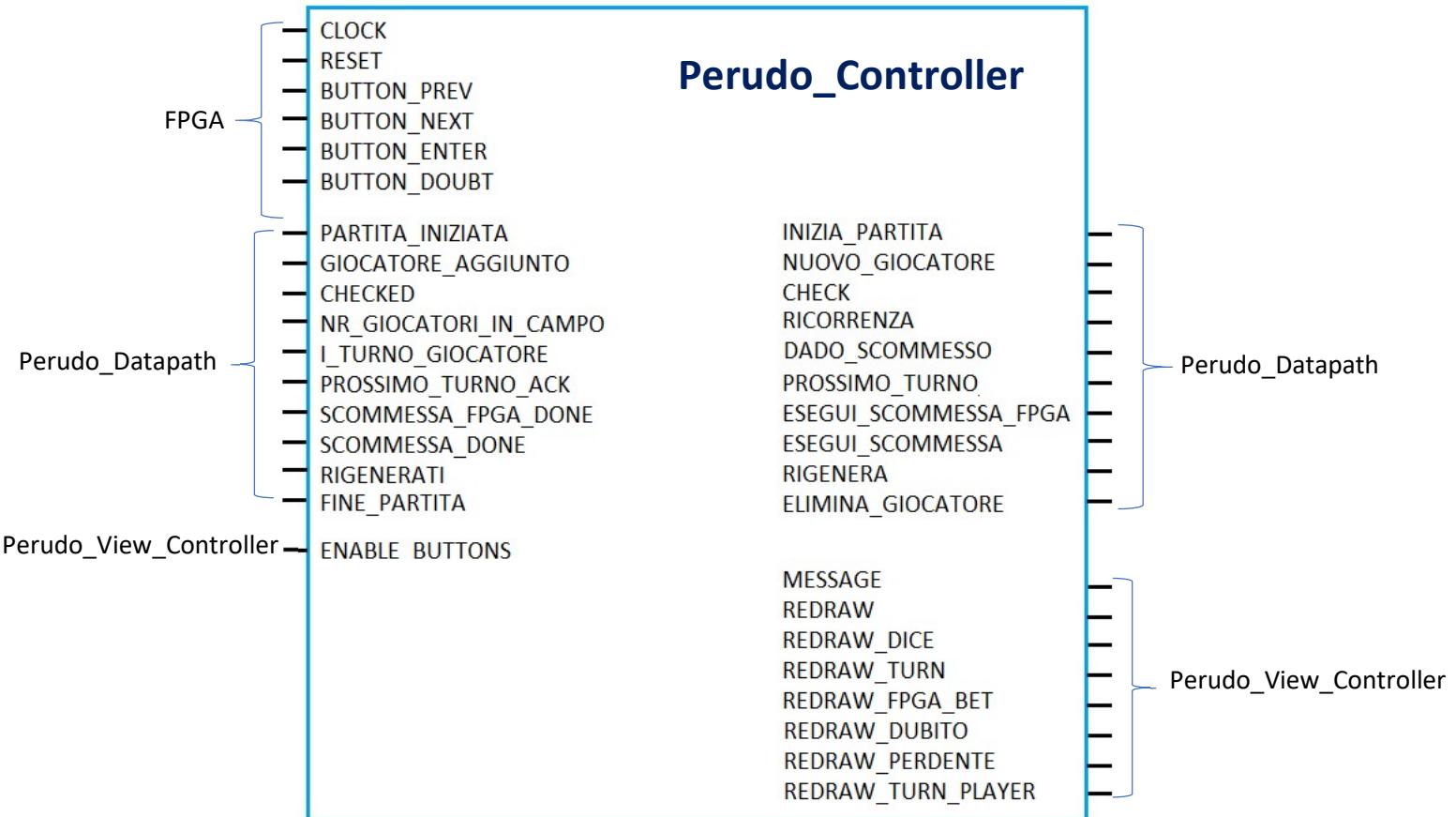


Fig 27: Macro-blocco del Perudo_Controller

Come mostrato in figura, il Perudo_Controller gestisce la pressione dei quattro pulsanti presenti sulla board FPGA e comunica con i componenti principali del sistema, ovvero il Perudo_Datapath e il Perudo_View_Controller.

La comunicazione con il Perudo_Datapath è realizzata attraverso un protocollo di Handshake in cui, per ogni segnale inviato dal Perudo_Controller, si aspetta un segnale di ACK da parte del Perudo_Datapath. Così facendo è possibile temporizzare tutte le operazioni e consentire all'utente di proseguire con la partita solo quando il sistema si trova in uno stato consistente.

La comunicazione con il Perudo_View_Controller avviene con la stessa logica descritta precedentemente, con la differenza che come unico segnale di ACK viene ricevuto il segnale ENABLE_BUTTONS, che viene utilizzato per consentire all'utente la pressione dei pulsanti.

8.1 GESTIONE DEI PULSANTI

Per consentire all'utente di giocare, scegliere la propria scommessa e avanzare con le fasi del gioco, sono stati utilizzati i quattro pulsanti presenti sulla board FPGA, i quali sono stati assegnati ai seguenti segnali:

BUTTON_PREV	=> not(KEY(3)),
BUTTON_NEXT	=> not(KEY(2)),
BUTTON_ENTER	=> not(KEY(1)),
BUTTON_DOUBT	=> not(KEY(0))

Fig 28: Definizione pulsanti

Per ovviare al problema che un click di un pulsante possa durare più di un singolo ciclo di clock e quindi possa generare un comportamento indesiderato, sono state utilizzate quattro variabili:

```
variable next_old : std_logic;
variable prev_old : std_logic;
variable enter_old : std_logic;
variable doubt_old : std_logic;
```

Fig 29: Definizione variabili associate ai pulsanti

Ad ogni ciclo di clock viene assegnato a ciascuna variabile il valore del relativo pulsante, così facendo è possibile associare un comportamento ad un click quando la relativa variabile ha valore '0' (come vedremo negli esempi successivi). In questo modo abbiamo la certezza che il pulsante resterà attivo per un solo ciclo di clock.

```
next_old := BUTTON_NEXT;
prev_old := BUTTON_PREV;
enter_old := BUTTON_ENTER;
doubt_old := BUTTON_DOUBT;
```

Fig 30: Assegnamento valore dei pulsanti

8.2 GESTIONE DEGLI STATI

La gestione degli stati avviene all'interno di un unico processo chiamato *Update_State_Controller*.

Gli stati principali sono 5: IDLE, INIT, TURN_PLAYER, TURN_FPGA, CHECK_WINNER, FINISH.

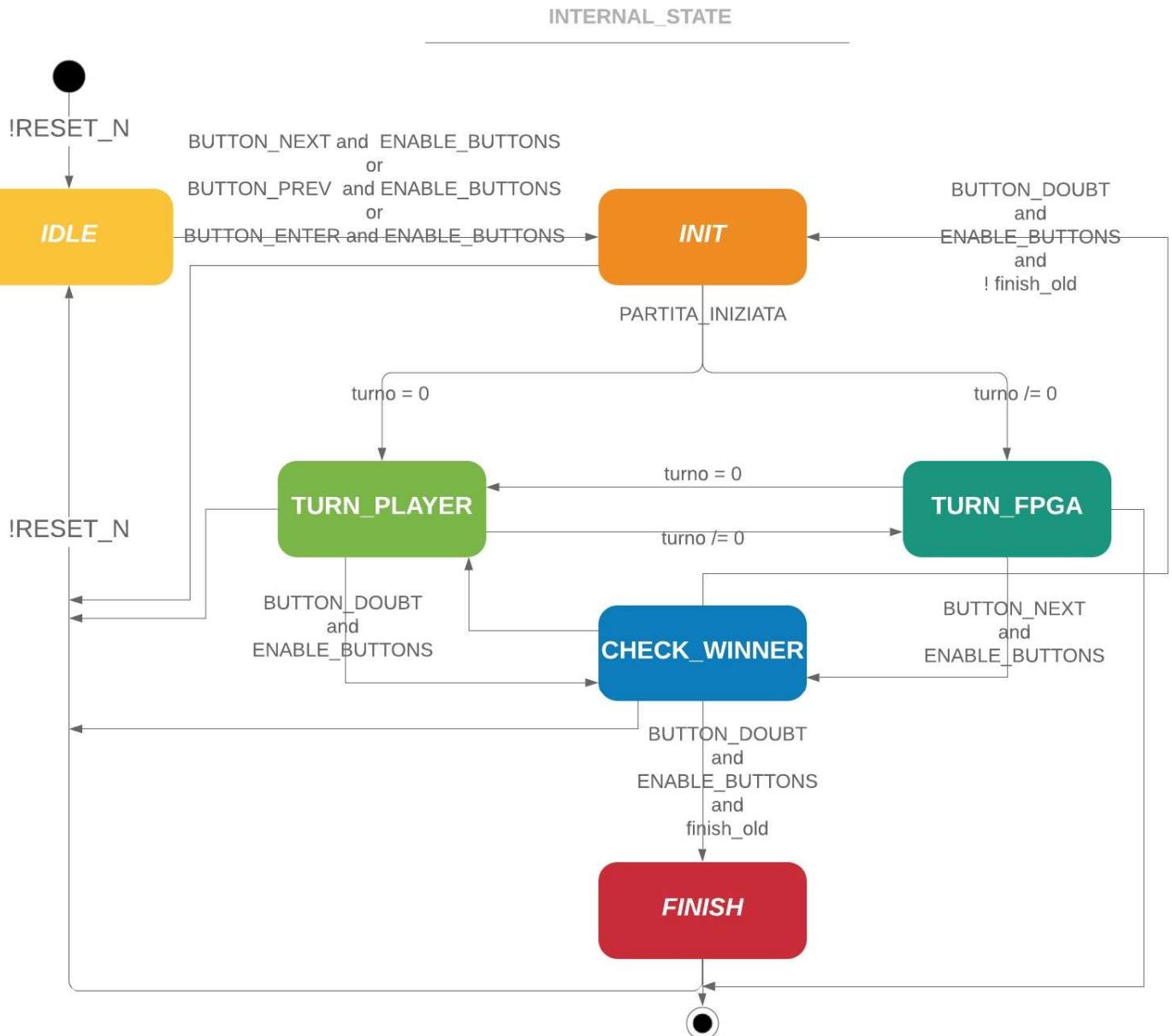


Fig 31: Diagramma degli stati della Control Unit

Grazie a questa gestione a stati, i quattro pulsanti che l'utente utilizzerà per giocare la partita assumeranno comportamenti diversi in base allo stato attuale del sistema.

IDLE

È il primo stato in cui si trova la variabile *internal_state* quando si resetta il sistema. Questo stato ha l'obiettivo di inizializzare tutti i segnali e tutte le variabili ad un valore prestabilito.

```
case (internal_state) is
    -----
    -- Fase IDLE
    -----
    when IDLE =>
        --Select number of players
        if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1') then
            send_start <= '1';
            internal_state <= INIT;
        elsif (BUTTON_PREV = '1' and prev_old = '0' and ENABLE_BUTTONS = '1') then
            send_start <= '1';
            internal_state <= INIT;
        elsif (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then
            send_start <= '1';
            internal_state <= INIT;
        elsif (BUTTON_DOUBT = '1' and doubt_old = '0' and ENABLE_BUTTONS = '1') then
            clean_buffer <= '1';
        end if;
```

Fig 32: Codice VHDL dello stato IDLE

Da questo stato è possibile passare allo stato di inizializzazione della partita con il click di un pulsante qualsiasi, fatta eccezione per il pulsante nominato BUTTON_DOUBT che viene utilizzato per la sincronizzazione della comunicazione seriale.

INIT

In questo stato avviene l'inizializzazione della partita, ovvero la scelta del proprio Avatar e la possibilità di scegliere il numero di giocatori (pilotati dall'FPGA) contro cui giocare.



Fig 33: Diagramma dei sottostati dello stato INIT

INIT prevede due sottostati: il primo (AVATAR) consente di utilizzare i pulsanti BUTTON_PREV e BUTTON_NEXT per la scelta dell'avatar con cui giocare (scelta puramente grafica che non viene mantenuta in memoria). Dopo di che, al click del pulsante BUTTON_ENTER, si conferma la scelta del proprio avatar e si passa alla selezione del numero di giocatori.

```

-----  

-- Fase di Inizializzazione  

-----  

when INIT =>  

  if(PARTITA_INIZIATA = '1') then
    if(turno = 0) then
      internal_state      <= TURN_PLAYER;
      turn_player_state   <= DADO;
    else
      internal_state      <= TURN_FPGA;
      turn_fpga_state     <= BET;
    end if;
  end if;  

  stato <= "01";  

  case (initialization_state) is  

  when AVATAR =>
    stato_interno <= "01";
    if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1') then
      -----
      -- Next Avatar
      -----
      increment_avatar  <= '1';
    elsif (BUTTON_PREV = '1' and prev_old = '0' and ENABLE_BUTTONS = '1') then
      -----
      -- Previous Avatar
      -----
      decrement_avatar <= '1';
    elsif (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then
      -----
      -- Vai alla selezione numero giocatori
      -----
      initialization_state <= NUM;
      NUOVO_GIOCATORE     <= '1';
      nuovo_giocatore_old := '1';
      select_avatar        <= '1';
    end if;

```

Fig 34: Codice VHDL del sottostato AVATAR dello stato INIT

Come si può notare dal codice soprastante, il primo controllo che viene effettuato riguarda il segnale PARTITA_INIZIATA ricevuto dal Perudo_Datapath; nel caso in cui questo avesse valore '1', *internal_state* passerà a TURN_FPGA o TURN_PLAYER in base al turno generato casualmente dal Perudo_Datapath stesso.

Una volta passati allo stato NUM sarà possibile utilizzare gli stessi pulsanti per incrementare o decrementare il numero di giocatori fino ad un minimo di 2 (numero minimo per giocare). Infine, con il pulsante BUTTON_ENTER, si invierà il segnale INIZIA_PARTITA al Perudo_Datapath

che, dopo aver aggiornato lo stato del sistema, risponderà con il segnale PARTITA_INIZIATA che consentirà al Perudo_Controller di passare allo stato successivo e dar inizio alla partita.

```

when NUM =>
    stato_interno <= "10";
    if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1'
        and nuovo_giocatore_old = '0' and numero_giocatori < 8) then
    -----
    -- Aggiungi un giocatore
    -----
    NUOVO_GIOCATORE      <= '1';
    nuovo_giocatore_old  := '1';

    elsif (BUTTON_PREV = '1' and prev_old = '0' and ENABLE_BUTTONS = '1'
           and elimina_giocatore_old = '0' and numero_giocatori > 1) then
    -----
    -- Rimuovi un giocatore
    -----
    ELIMINA_GIOCATORE     <= '1';
    elimina_giocatore_old := '1';

    elsif (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then
    -----
    -- Inizia la partita
    -----
    INIZIA_PARTITA          <= '1';
    inizia_partita_old      := '1';
    initialization_state    <= AVATAR;

    end if;
end case;

```

Fig 35: Codice VHDL del sottostato NUM dello stato INIT

TURN_PLAYER

Questo stato consente all’utente di scegliere la propria scommessa basandosi su una scommessa effettuata in precedenza da un qualsiasi altro giocatore. Questo turno è considerato dal sistema come il turno 0, in quanto è il primo giocatore che viene inizializzato in una partita. Di conseguenza, ogni volta che il Perudo_Controller riceverà il segnale turno = 0, passerà a questo stato.

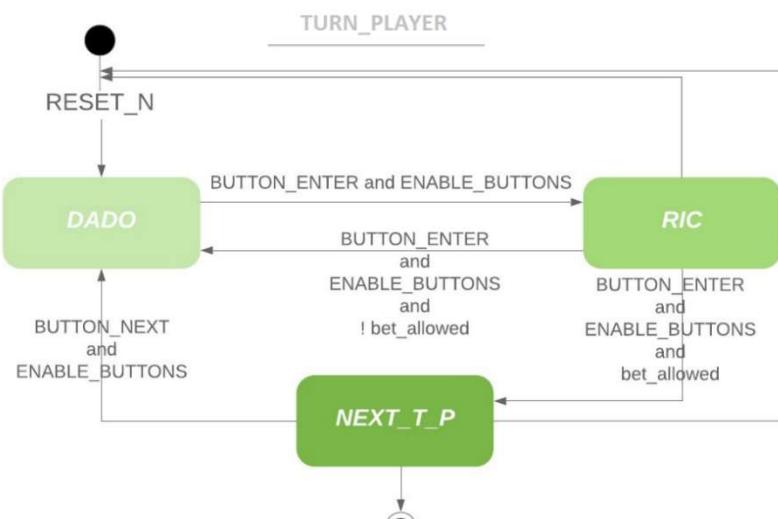


Fig 36: Diagramma dei sottostati dello stato TURN_PLAYER

Come per lo stato di INIT, anche lo stato TURN_PLAYER contiene tre sottostati: il primo (DADO) consente di scegliere, utilizzando i soliti pulsanti BUTTON_PREV e BUTTON_NEXT, la faccia del dado che si vuole scommettere, mentre, con il pulsante BUTTON_ENTER si conferma il dado e si passa alla selezione della ricorrenza.

```
-- Fase di gioco del Giocatore
-----
when TURN_PLAYER =>

  if(PARTITA_INIZIATA = '1') then
    if(turno /= 0) then
      internal_state      <= TURN_FPGA;
      turn_fpga_state     <= BET;
    end if;
  end if;

  stato <= "10";

  case (turn_player_state) is

    when DADO =>
      stato_interno <= "01";
      if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1' and dado_temp < 6) then
        -----
        -- Incrementa la faccia del dado
        -----
        dado_temp          <= dado_temp + 1;
        increment_bet     <= '1';

      elsif (BUTTON_PREV = '1' and prev_old = '0' and ENABLE_BUTTONS = '1' and dado_temp > 1) then
        -----
        -- Decrementa la faccia del dado
        -----
        dado_temp          <= dado_temp - 1;
        decrement_bet     <= '1';

      elsif (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then
        -----
        -- Conferma la faccia del dado
        -----
        turn_player_state  <= RIC;
        select_bet         <= '1';

      elsif (BUTTON_DOUBT = '1' and doubt_old = '0' and ENABLE_BUTTONS = '1' and check_old = '0') then
        -----
        -- Dubita la scommessa
        -----
        CHECK              <= '1';
        check_old          := '1';
        turn_player_state <= DADO;
      end if;
    end case;
  end if;
end when;
```

Fig 37: Codice VHDL del sottostato DADO dello stato TURN_PLAYER

Come in precedenza, viene controllato per prima cosa il turno del giocatore, se si riceve un turno diverso da 0 si passa direttamente alla fase TURN_FPGA, altrimenti si processano i sottostati di TURN_PLAYER.

Dopo aver confermato la scelta della faccia del dado, ci si trova nello stato RIC in cui, con i medesimi pulsanti, si può selezionare e confermare la ricorrenza del dado che si vuole scommettere. Dopo aver selezionato pure la ricorrenza si conferma la scommessa. Tuttavia, l'utente potrebbe fare una puntata errata per le regole del Perudo (ad esempio potrebbe scommettere lo stesso dado scommesso in precedenza però con una ricorrenza minore); per risolvere questo problema, viene utilizzata una funzione del Perudo_Package che consente di verificare se i dati che si vogliono puntare sono validi oppure no: in caso positivo si procede con il salvataggio della nuova scommessa passando al sottostato NEXT_T_P, altrimenti si torna allo stato di selezione del dado (DADO).

```

when RIC =>
    stato_interno <= "10";
    if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1' and ricorrenza_temp < 40) then
        -----
        -- Incrementa la ricorrenza
        -----
        ricorrenza_temp      <= ricorrenza_temp + 1;
        increment_bet        <= '1';
    elsif (BUTTON_PREV = '1' and prev_old = '0' and ENABLE_BUTTONS = '1' and ricorrenza_temp > 1) then
        -----
        -- Decrementa la ricorrenza
        -----
        ricorrenza_temp      <= ricorrenza_temp - 1;
        decrement_bet        <= '1';
    elsif (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1' and check_old = '0') then
        --
        -- Conferma la scommessa
        --
        if (bet_allowed = '1') then
            DADO_SCOMMESSO  <= converti_da_intero_a_dado(dado_temp);
            RICORRENZA       <= ricorrenza_temp;
            ESEGUI_SCOMMESSA <= '1';
            turn_player_state <= NEXT_T_P;
        else
            send_error      <= '1';
            turn_player_state <= DADO;
        end if;
    elsif (BUTTON_DOUBT = '1' and doubt_old = '0' and ENABLE_BUTTONS = '1' ) then
        CHECK           <= '1';
        check_old       := '1';
        turn_player_state <= DADO;
    end if;
}

```

Fig 38: Codice VHDL del sottostato RIC dello stato TURN_PLAYER

Nello stato DADO, così come nello stato RIC, è possibile utilizzare il pulsante BUTTON_DOUBT per dubitare la scommessa dell'avversario precedente, inviare il segnale di CHECK al Perudo_Datapath e passare allo stato principale CHECK_WINNER.

Un'altra cosa che si può notare dal codice è che sono stati inseriti dei limiti sia per la scelta della faccia del dado che per la scelta della ricorrenza; in particolare sarà possibile incrementare e decrementare la faccia del dado in un range da 1 a 6, mentre la scelta della ricorrenza sarà limitata da un range da 1 a 40 (numero massimo di dadi possibili in campo con 8 giocatori).

L'ultimo sottostato di TURN_PLAYER è NEXT_T_P, che consiste in un semplice stato di transizione al prossimo turno, utilizzato principalmente per temporizzare tutte le fasi della partita con il click dei pulsanti.

```

when NEXT_T_P =>
    stato_interno <= "11";
    if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1') then
        -----
        -- Vai al prossimo turno
        -----
        PROSSIMO_TURNO    <= '1';
        turn_player_state <= DADO;
    end if;
end case;

```

Fig 39: Codice VHDL del sottostato NEXT_T_P dello stato TURN_PLAYER

TURN_FPGA

Lo stato TURN_FPGA gestisce la fase di gioco di un giocatore pilotato dall'FPGA.

Questa fase può essere considerata come l'equivalente della fase TURN_PLAYER, dove però la scommessa non viene scelta dall'utente, bensì viene generata automaticamente dall'AI_Controller.

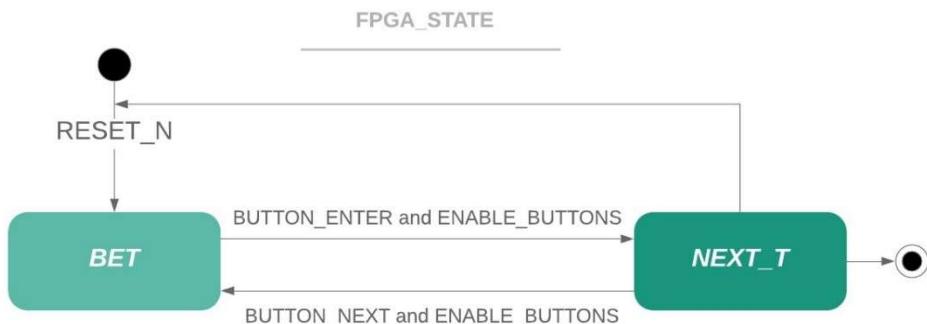


Fig 40: Diagramma dei sottostati dello stato TURN_FPGA

Anche questa fase prevede due sottostati: BET e NEXT_T.

La fase di BET consente la pressione di un unico pulsante (BUTTON_ENTER), che consiste nel richiedere la generazione della scommessa automatica da parte dell'AI_Controller.

Successivamente, dopo aver ricevuto i vari ACK da parte del Perudo_Datapath che confermano la generazione della scommessa, si passa alla fase NEXT_T che, esattamente come la fase NEXT_T_P dello stato TURN_PLAYER, consente la transizione temporizzata al turno del prossimo giocatore tramite il click del pulsante BUTTON_NEXT.

```

-----  

-- Fase di gioco dell'FPGA  

-----  

when TURN_FPGA =>  

  if(PARTITA_INIZIATA = '1') then  

    if(turno = 0) then  

      internal_state      <= TURN_PLAYER;  

      turn_player_state   <= DADO;  

    end if;  

  end if;  

  stato <= "11";  

  case (turn_fpga_state) is  

    when BET =>  

      stato_interno <= "01";  

      if (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then  

        -----  

        -- Genera la scommessa del giocatore  

        -----  

        ESEGUI_SCOMMESSA_FPGA  <= '1';  

        DAMMI_TURNO_GIOCATORE  <= '1';  

        turn_fpga_state        <= NEXT_T;  

      end if;  

    when NEXT_T =>  

      stato_interno <= "10";  

      if (BUTTON_NEXT = '1' and next_old = '0' and ENABLE_BUTTONS = '1' and check_old = '0') then  

        -----  

        -- Vai al prossimo turno  

        -----  

        PROSSIMO_TURNO       <= '1';  

        turn_fpga_state       <= BET;  

      end if;  

  end case;

```

Fig 41: Codice VHDL dello stato TURN_FPGA

Come per INIT e TURN_PLAYER viene controllato il turno della partita, per passare eventualmente allo stato TURN_PLAYER. Una volta generata la scommessa e cliccato il pulsante NEXT_T viene impostato a '1' il segnale PROSSIMO_TURNO, che indica al Datapath di incrementare il turno del giocatore. Se il turno del giocatore rimane diverso da zero, significa che il prossimo giocatore è ancora un giocatore pilotato dall'FPGA, quindi si dovrà rimanere nello stato attuale, altrimenti si passa allo stato TURN_PLAYER.

CHECK_WINNER

Questo stato viene raggiunto solo in due casi: quando l'utente, durante la fase TURN_PLAYER, clicca il pulsante BUTTON_DOUBT, oppure quando la scommessa generata dall'AI_Controller durante la fase TURN_FPGA corrisponde ad un DUBITO.

```

if(SCOMMESSA_FPGA_DONE = '1') then
  if(SCOMMESSA_ATTUALE.dado_scommesso = NONE) then
    -----  

    -- Dubito  

    -----  

    CHECK                  <= '1';
    check_old              := '1';
    turn_fpga_state        <= BET;
  else
    confirm_fpga_bet      <= '1';
  end if;

```



NONE viene considerato dal sistema come un DUBITO

Fig 42: Codice VHDL di una scommessa generata dall'AI_Controller corrispondente ad un DUBITO

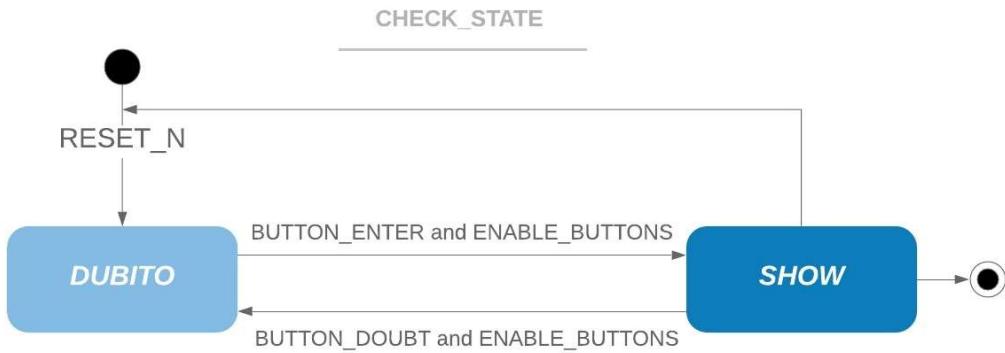


Fig 43: Diagramma dei sottostati dello stato CHECK_WINNER

Questa fase contiene le sottofasi: DUBITO e SHOW.

Queste sottofasi sono state introdotte principalmente con l'obiettivo di temporizzare gli elementi da mostrare all'utente. Dunque il loro scopo è quello di inviare al Perudo_View_Controller le informazioni necessarie affinché, durante la fase DUBITO, venga mostrato il nome del giocatore che ha perso la scommessa, mentre durante la fase SHOW vengano rigenerati i dadi dei giocatori e venga inizializzato un nuovo round.

```

-- Fase di controllo dadi
-----
when CHECK_WINNER =>
  case (check_state) is
    when DUBITO =>
      stato_interno <= "01";
      if (BUTTON_ENTER = '1' and enter_old = '0' and ENABLE_BUTTONS = '1') then
        -- Invia dati del perdente alla View
        check_state      <= SHOW;
        send_perdente   <= '1';
      end if;
    when SHOW =>
      stato_interno <= "10";
      if (BUTTON_DOUBT = '1' and doubt_old = '0' and ENABLE_BUTTONS = '1') then
        if(finish_old = '1') then
          -- Partita terminata
          send_finish      <= '1';
          internal_state   <= FINISH;
        else
          -- Nuovo Round
          RIGENERA           <= '1';
          internal_state     <= INIT;
          check_state        <= DUBITO;
        end if;
      end if;
  end case;

```

```

if (FINE_PARTITA = '1') then
  finish_old           <= '1';
end if;

```

Fig 44: Codice VHDL dello stato CHECK_WINNER

8.3 INTERAZIONE CON PERUDO_VIEW_CONTROLLER

L’interazione con il Perudo_View_Controller è gestita da un processo separato che, in base ai segnali interni, richiede l’invio di determinati dati. Questi si basano su un protocollo di comunicazione stabilito a priori per le interazioni tra l’FPGA e la parte grafica realizzata in Unity.

```

if(clean_buffer = '1') then
    MESSAGE  <= "01010101";
    REDRAW   <= '1';
end if;

if(send_start = '1') then
    MESSAGE  <= "01001001"; -- 73 => 'I'
    REDRAW   <= '1';
end if;

if(increment_avatar = '1') then
    MESSAGE  <= "01000100"; --68 => 'D'
    REDRAW   <= '1';
end if;

if(decrement_avatar = '1') then
    MESSAGE  <= "01010011"; -- 83 => 'S'
    REDRAW   <= '1';
end if;

if(select_avatar = '1' ) then
    MESSAGE  <= "01001111"; -- 79 => 'O'
    REDRAW   <= '1';
end if;

if(send_dice = '1') then
    REDRAW_DICE <= '1';
end if;

if(send_dubito = '1') then
    REDRAW_DUBITO <= '1';
end if;

if(send_perdente = '1') then
    REDRAW_PERDENTE  <= '1';
end if;

if(select_bet = '1') then
    MESSAGE  <= "01001111"; -- 79 => 'O'
    REDRAW   <= '1';
end if;

if(send_error = '1') then
    MESSAGE  <= "01100101"; -- 101 => 'e'
    REDRAW   <= '1';
end if;

if(increment_bet = '1') then
    MESSAGE  <= "01000100"; --68 => 'D'
    REDRAW   <= '1';
end if;

if(decrement_bet = '1') then
    MESSAGE  <= "01010011"; -- 83 => 'S'
    REDRAW   <= '1';
end if;

if(send_finish = '1') then
    MESSAGE  <= "01011001"; -- 89 => 'Y'
    REDRAW   <= '1';
end if;

if(confirm_bet = '1') then
    REDRAW_TURN_PLAYER  <= '1';
end if;

if(confirm_fpga_bet = '1') then
    REDRAW_FPGA_BET    <= '1';
end if;

```

Fig 45: Codice VHDL della generazione dei segnali di interazione con il Perudo_View_Controller

Nell’immagine sono rappresentati tutti i segnali utilizzati per implementare il protocollo di comunicazione gestito poi dal Perudo_View_Controller.

Per ogni segnale interno impostato a 1 all’interno del processo di gestione degli stati, viene inviato un segnale apposito al Perudo_View_Controller, che sarà in grado di interpretarlo e inviare le corrette informazioni attraverso il canale seriale.

Si può osservare dal codice che quando il messaggio da inviare è un singolo Byte, è il Perudo_Controller stesso a specificarlo, mentre quando la comunicazione è più complessa e necessita di inviare più Byte, sarà il Perudo_View_Controller (come vedremo nel paragrafo successivo) ad occuparsene.

Tutti i segnali presenti nel codice soprastante servono solo a notificare determinati eventi, di conseguenza verranno resettati a 0 ad ogni ciclo di clock.

9 PERUDO_VIEW_CONTROLLER

Il Perudo_View_Controller si occupa della gestione del protocollo di comunicazione dei dati e della trasmissione degli stessi attraverso il canale seriale.

È composto da un processo principale e un componente MY_TX.

Quest'ultimo si occupa dell'invio dei singoli byte attraverso il canale seriale, gestendo tutte le complessità derivanti da questa operazione (come vedremo nel paragrafo dedicato). Il processo principale si occupa invece di inserire nel canale di comunicazione di MY_TX i messaggi che si vogliono trasmettere in base allo stato attuale del sistema.

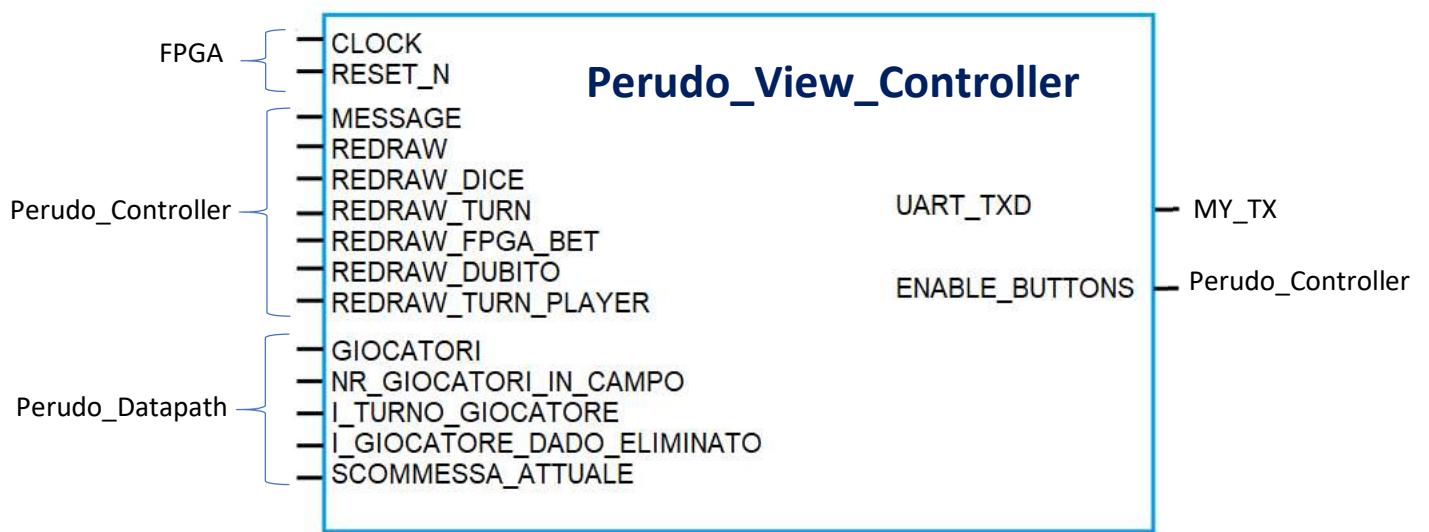


Fig 46: Macro-blocco del Perudo_View_Controller

I segnali ricevuti in ingresso dal Perudo_Controller notificano che tipo di dato è necessario trasmettere attraverso la seriale, mentre quelli ricevuti dal Perudo_Datapath specificano il dato vero e proprio che andrà inserito nel canale di comunicazione.

In uscita sono presenti solo due segnali: il primo (UART_RXD) è il segnale di trasmissione che verrà utilizzato all'interno del componente MY_TX per occupare il canale, mentre ENABLE_BUTTONS viene impostato a '1' ogni volta che il canale di comunicazione si libera (ovvero non appena sono stati trasmessi tutti i dati alla grafica Unity correttamente). Così facendo consentiamo la pressione dei pulsanti solo quando ci troviamo in uno stato stabile del sistema.

9.1 MY_TX

Come anticipato, il componente MY_TX si occupa della trasmissione vera e propria dei byte sul canale di comunicazione seriale.

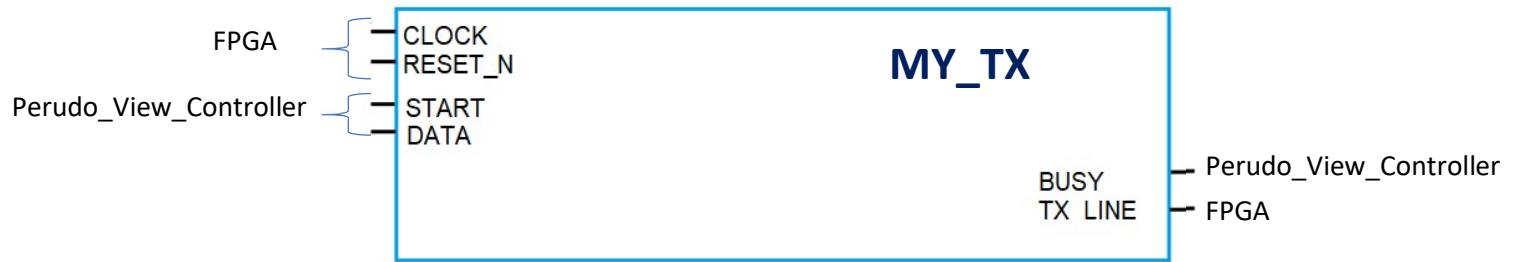


Fig 47: Macro-blocco di MY_TX

START e DATA sono i due segnali in ingresso generati internamente dal Perudo_View_Controller.

START notifica l'intenzione di voler iniziare una trasmissione, mentre DATA è un vettore di 8 bit che indica il byte che si andrà ad inviare nel canale.

Per quanto riguarda i segnali di uscita, BUSY notifica al Perudo_View_Controller se il canale è attualmente occupato, mentre TX_LINE è associato al canale fisico dell'FPGA.

```

process (CLK, START)

begin
if(rising_edge(CLK) ) then
    if(TX_FLG='0' and START='1') then
        TX_FLG<='1';
        BUSY<='1';
        DATAFLL(0)<='0'; --Start bit
        DATAFLL(9)<='1'; --End bit
        DATAFLL(8 downto 1)<=DATA;
    end if;
    if(TX_FLG='1') then
        if(PRSCL<5207) then
            PRSCL<=PRSCL+1;
        else
            PRSCL<=0;
        end if;
        if(PRSCL = 2607) then
            -----
            -- Trovata la metà del ciclo di clock
            -----
            TX_LINE<=DATAFLL(INDEX);
            if(INDEX<9) then
                INDEX<=INDEX+1;
            else
                TX_FLG<='0';
                BUSY<='0';
                INDEX<=0;
            end if;
            end if;
        end if;
    end if;
end process;
    
```

Fig 48: Codice VHDL del componente UART TX

MY_TX segue la linea di un Universal Receiver Transmitter o UART. Il trasmettitore prende un byte alla volta e lo trasmette in maniera sequenziale. La linea sulla quale vengono trasmessi i bit è di norma a valore logico alto, per questo motivo ogni byte è preceduto da un bit a 0 che indica l'inizio di una trasmissione.

MY_TX utilizza il CLOCK a 50 MHz della board FPGA, mentre la velocità di trasmissione attraverso la porta seriale è di 9600b/s, cioè Baud Rate = 9600. È possibile notare che un bit sarà valido per un certo numero di clock della FPGA.

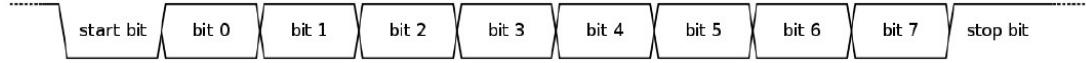


Fig 49: Sequenza di bit di trasmissione di MY_TX

Al reset della board, il transmitter si sincronizza con il receiver inviando 1 o più byte di sincronizzazione. Per questo motivo è stata lasciata la possibilità di svuotare il canale attraverso il pulsante BUTTON_DOUBT durante la fase IDLE del Perudo_Controller.

9.2 IL PROTOCOLLO DI COMUNICAZIONE

Come nel Perudo_Controller, anche nel Perudo_View_Controller viene utilizzata una gestione a stati.

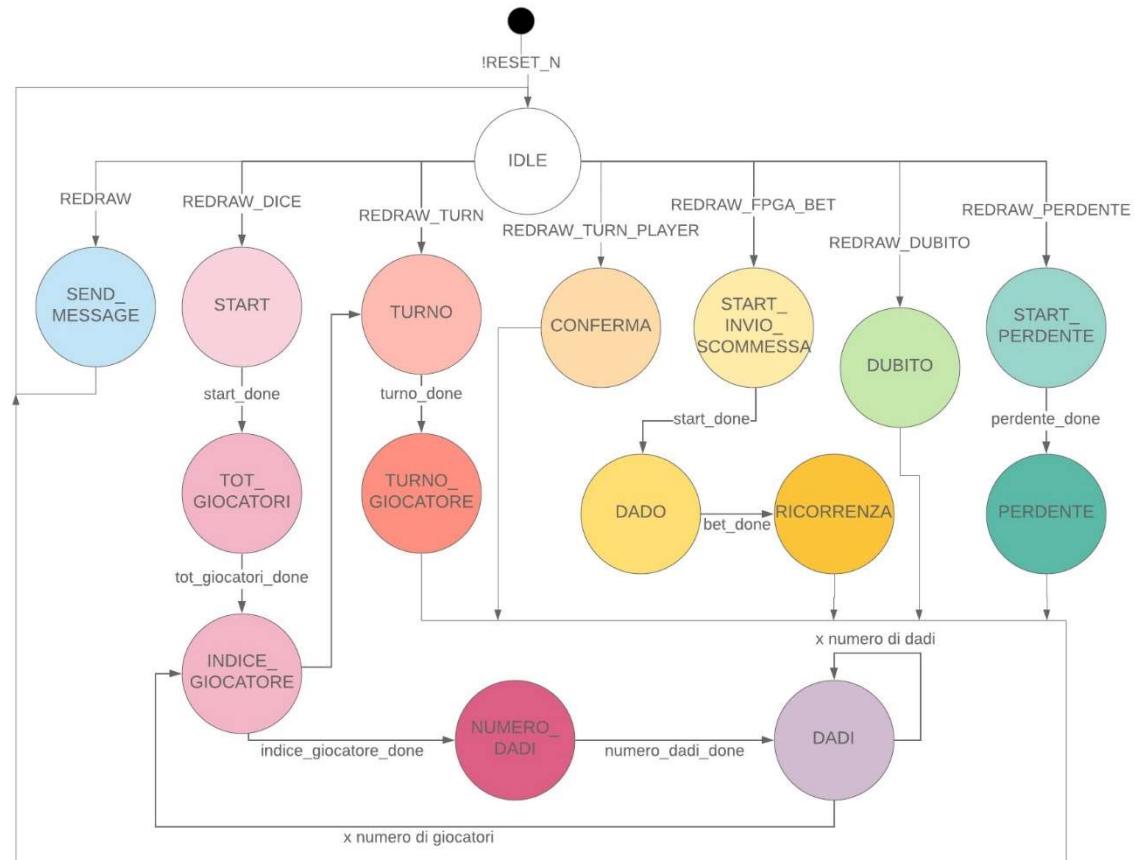


Fig 50: Diagramma degli stati del Perudo_View_Controller

Ogni stato si occupa di inviare un byte seguendo il protocollo di interazione con la View che verrà spiegato più in dettaglio nel paragrafo successivo.

IDLE

Al reset della board, il componente si trova nello stato IDLE in attesa di ricevere segnali dal Perudo_Controller.

```

case (tx_state) is
when IDLE =>
    ENABLE_BUTTONS      <= '1';
-----
-- REDRAW VIEW
-----
if (REDRAW = '1' and TX_BUSY='0') then
    tx_state           <= SEND_MESSAGE;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND DICE
-----
if (REDRAW_DICE = '1' and TX_BUSY='0') then
    tx_state           <= START;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND NEW TURN
-----
if (REDRAW_TURN = '1' and TX_BUSY='0') then
    tx_state           <= TURNO;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND NEXT TURN AFTER PLAYER 0
-----
if (REDRAW_TURN_PLAYER = '1' and TX_BUSY='0') then
    tx_state           <= CONFERMA;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND FPGA BET
-----
if (REDRAW_FPGA_BET = '1' and TX_BUSY='0') then
    tx_state           <= START_INVIO_SCOMMESSA;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND DUBITO BET
-----
if (REDRAW_DUBITO = '1' and TX_BUSY='0') then
    tx_state           <= DUBITO;
    ENABLE_BUTTONS     <= '0';
end if;
-----
-- SEND PERDENTE
-----
if (REDRAW_PERDENTE = '1' and TX_BUSY='0') then
    tx_state           <= START_PERDENTE;
    ENABLE_BUTTONS     <= '0';
end if;

```

Fig 51: Codice VHDL dello strato IDLE del Perudo_View_Controller

Appena viene ricevuto un segnale dal Perudo_Controller si passa ad uno degli stati gestiti e viene disabilitato il click dei pulsanti ponendo ENABLE_BUTTONS a ‘0’.

SEND_MESSAGE

Questo stato è il più semplice in quanto gestisce l’invio di un unico byte. È l’unico stato che manda direttamente il dato contenuto all’interno del segnale MESSAGE inviato dal Perudo_Controller.

```

when SEND_MESSAGE =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= MESSAGE(7 downto 0);
      TX_START         <= '1';
      start_done       := '1';
      ready            := '0';
    else
      ready := '1';
    end if;
  else
    TX_START        <= '0';
    if(start_done = '1') then
      tx_state        <= IDLE;
      start_done      := '0';
    end if;
  end if;

```

Fig 52: Codice VHDL dello stato SEND_MESSAGE del Perudo_View_Controller

START

Lo stato START indica l’inizio della trasmissione di tutti i dati da mostrare in una scena di gioco. Questa fase è la più complessa in quanto sarà necessario fornire alla View le informazioni riguardanti tutti i dadi posseduti da ciascun giocatore ancora in campo.

Per prima cosa viene inviata la lettera ‘A’ per segnalare alla View di prepararsi a ricevere i dati relativi ai dadi dei giocatori.

```

when START =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= "01000001"; -- 65 -> A
      TX_START         <= '1';
      start_done       := '1';
      ready            := '0';
    else
      ready := '1';
    end if;
  else
    TX_START        <= '0';
    if(start_done = '1') then
      tx_state        <= TOT_GIOCATORI;
      start_done      := '0';
    end if;
  end if;

```

Fig 53: Codice VHDL dello stato START del Perudo_View_Controller

Una volta inviata la lettera ‘A’ si passa allo stato TOT_GIOCATORI, in cui si invia il numero di giocatori in campo ricevuto in ingresso direttamente dal Perudo_Datapath.

```
when TOT_GIOCATORI =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= std_logic_vector(to_signed((NR_GIOCATORI_IN_CAMPO + 48),8));
      TX_START         <= '1';
      tot_giocatori_done := '1';
      ready            := '0';
    else
      ready := '1';
    end if;
  else
    TX_START     <= '0';
    if(tot_giocatori_done = '1') then
      tx_state       <= INDICE_GIOCATORE;
      tot_giocatori_done := '0';
    end if;
  end if;
```

Fig 54: Codice VHDL dello stato TOT_GIOCATORI del Perudo_View_Controller

Da notare che l'intero inviato viene sommato al numero 48 affinché il carattere che verrà ricevuto corrisponda al codice ASCII del corrispettivo numero intero.

Successivamente si passa alla fase INDICE_GIOCATORE in cui si invia l'indice del giocatore del quale si stanno per inviare i dati.

```
when INDICE_GIOCATORE =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      if(indice < MAX_GIOCATORI and GIOCATORI(indice).dadi_in_mano(0) /= NONE
        and GIOCATORI(indice).numero_dadi_in_mano > 0 ) then
        TX_DATA          <= std_logic_vector(to_signed((indice+48),8));
        TX_START         <= '1';
        indice_giocatore_done := '1';
        ready            := '0';
      else
        if(indice = MAX_GIOCATORI) then
          indice := 0;
          tx_state <= TURNO;
        else
          indice := indice + 1;
        end if;
      end if;
    else
      ready := '1';
    end if;
  else
    TX_START     <= '0';
    if(indice_giocatore_done = '1') then
      tx_state       <= NUMERO_DADI;
      indice_giocatore_done := '0';
    end if;
  end if;
```

```
variable indice : integer range 0 to 8 := 0;
begin
  if (RESET_N = '0') then
    ;
    indice           := 0;
    ;
end;
```

Fig 55: Codice VHDL dello stato INDICE_GIOCATORE del Perudo_View_Controller

In questa fase si può notare come, grazie al controllo inserito prima dell'invio dei dati, vengano considerati solo i giocatori in campo che possiedono almeno un dado.

Una volta inviato l'indice del giocatore si passa allo stato NUMERO_DADI in cui viene inviato il numero di dadi attualmente posseduti da quel giocatore.

```

when NUMERO_DADI =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= std_logic_vector(to_signed((GIOCATORI(indice).numero_dadi_in_mano+ 48),8));
      TX_START         <= '1';
      numero_dadi_done := '1';
      ready            := '0';
    else
      ready           := '1';
    end if;
  else
    TX_START         <= '0';
    if(numero_dadi_done = '1') then
      tx_state        <= DADI;
      numero_dadi_done := '0';
    end if;
  end if;
end if;

```

Fig 56: Codice VHDL dello stato NUMERO_DADI del Perudo_View_Controller

Finalmente si passa alla fase DADI che si occupa di inviare, in maniera sequenziale, tutti i dadi posseduti da quel giocatore.

```

when DADI =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      if(indice_dado < MAX_DADI and indice_dado < GIOCATORI(indice).numero_dadi_in_mano and
         GIOCATORI(indice).dadi_in_mano(indice_dado) /= NONE) then
        dado_int          := (converti_da_dado_a_intero(GIOCATORI(indice).dadi_in_mano(indice_dado)));
        TX_DATA          <= std_logic_vector(to_signed( dado_int + 48),8));
        TX_START         <= '1';
        dadi_done        := '1';
        ready            := '0';
      else
        indice          := indice + 1;
        indice_dado     := 0;
        tx_state        <= INDICE_GIOCATORE;
      end if;
    else
      ready           := '1';
    end if;
  else
    TX_START         <= '0';
    if(dadi_done = '1') then
      indice_dado     := indice_dado + 1;
      dadi_done       := '0';
    end if;
  end if;
end if;

```

Fig 57: Codice VHDL dello stato DADI del Perudo_View_Controller

Perudo_View_Controller rimarrà in questo stato fintanto che non avrà inviato tutti i dadi relativi al giocatore che si sta processando; dopo di che verrà incrementata la variabile *indice* e si tornerà allo stato INDICE_GIOCATORE per processare il giocatore successivo. Quando non ci saranno più dati da inviare si passerà allo stato TURNO.

TURNO

In questo stato viene inviato il turno del giocatore che deve effettuare la scommessa, quindi verrà inviata la lettera ‘T’ alla View per notificare la fine dell’invio dei dadi e l’inizio dell’invio del turno.

```
when TURNO =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= "01010100"; --      T -> 84
      TX_START         <= '1';
      turno_done      := '1';
      ready            := '0';
    else
      ready           := '1';
    end if;
  else
    TX_START         <= '0';
    if(turno_done = '1') then
      turno_done     := '0';
      tx_state       <= TURNO_GIOCATORE;
    end if;
  end if;
```

Fig 58: Codice VHDL dello stato TURNO del Perudo_View_Controller

Successivamente si passa alla fase TURNO_GIOCATORE in cui si invia l’indice del giocatore ricevuto grazie al Perudo_Datapath.

```
when TURNO_GIOCATORE =>
  if(TX_BUSY = '0') then
    if(ready = '1') then
      TX_DATA          <= std_logic_vector(to_signed(I_TURNO_GIOCATORE + 48,8));
      TX_START         <= '1';
      turno_done      := '1';
      ready            := '0';
    else
      ready           := '1';
    end if;
  else
    TX_START         <= '0';
    if(turno_done = '1') then
      turno_done     := '0';
      tx_state       <= IDLE;
    end if;
  end if;
```

Fig 59: Codice VHDL dello stato TURNO_GIOCATORE del Perudo_View_Controller

CONFERMA

Lo stato CONFERMA viene raggiunto quando l’utente clicca sul pulsante BUTTON_ENTER per confermare la scelta dell’avatar, del dado o della ricorrenza. In questo caso viene inviata alla View una lettera ‘O’ con lo stesso principio con cui viene inviata la lettera ‘T’ durante la fase TURNO. Non essendoci altri byte da inviare si torna subito allo stato IDLE.

START_INVIO_SCOMMESSA

Questo stato viene raggiunto quando, dallo stato IDLE, si riceve REDRAW_FPGA_BET, ovvero quando il Perudo_Controller riceve una scommessa generata dall'AI_Controller con dado diverso da NONE. In questo caso i byte da inviare saranno tre: una ‘B’ per indicare alla View che stiamo inviando i dati di una scommessa e due interi: il primo indicante il dado scommesso mentre il secondo la ricorrenza.

DUBITO

Lo stato DUBITO, come lo stato CONFERMA, invierà un solo byte, questa volta corrispondente alla lettera ‘D’.

START_PERDENTE

Infine, lo stato START_PERDENTE serve a notificare l’indice del giocatore che ha perso la scommessa e quindi il dado a seguito di un DUBITO. In questo caso i byte da inviare saranno due: una ‘Z’ seguita da un intero.

10 VIEW

Come accennato in precedenza, l’interfaccia grafica è stata realizzata sfruttando il motore grafico Unity.

10.1 L’AMBIENTE UNITY

Al fine di garantire un’esperienza di gioco unica e travolgente, si è deciso di immergere l’utente in un mondo tridimensionale, in cui possa selezionare tra tanti, il personaggio che più gli piace e competere con altri giocatori nell’avvincente scontro del Perudo. Dovendo gestire e manipolare interattivamente oggetti virtuali, risulta essenziale la scelta di un opportuno motore grafico.

Unity, piattaforma notoriamente utilizzata per la realizzazione di ambienti tridimensionali (logo in Fig.60), implica l’utilizzo di programmazione a oggetti, supportando tre diversi linguaggi, C#, Javascript e Boo, anche se C# è quello mediamente più utilizzato. Si tratta di un motore grafico e in quanto tale facilita lo sviluppo di applicazioni offrendo librerie con funzioni, classi e dati già definiti senza dover programmare tutto quanto da zero. Originariamente le piattaforme di gioco erano software estremamente costosi, ma Unity scaricabile gratuitamente, ha segnato un punto di svolta costituendo un’importante innovazione nella comunità scientifica.

Per aggiungere interattività tra i vari oggetti creati in un progetto è necessario ricorrere a degli script e a tal fine può essere comodo utilizzare Visual Studio come ambiente di sviluppo.



Fig 60: Logo di Unity

10.2 PERUDO IN UNITY

Il gioco è suddiviso in sette scene principali, aggiornate e modificate a runtime a seconda dei segnali ricevuti in input dal canale seriale (vedi paragrafo 10.3).

All'avvio viene mostrata una schermata di benvenuto (Fig.61) che presenta il titolo del gioco ed alcuni dei personaggi principali, invitando l'utente a premere un qualunque tasto per iniziare la partita.



Fig 61: Scena 1, Schermata Iniziale

La seconda scena mostra il luogo in cui si affronteranno i giocatori: un'ampia sala accogliente su due piani, provvista di un bar, un arredamento un po' fuori dagli schemi in stile eccentrico, un barista instancabile e una zanzara fastidiosa (Fig.62). In questa fase l'utente può selezionare il suo personaggio preferito tra otto possibili.



Fig 62: Scena 2, Selezione Personaggio

Una volta confermata la scelta, ci si posiziona presso il tavolo da gioco e si decide quanti sfidanti affrontare, aggiungendoli uno ad uno, per un massimo di otto giocatori. Ognuno ha di fronte a sé il proprio bicchiere in cui lancerà i dadi per tenerli nascosti durante le scommesse (Fig.63).



Fig 63: Scena 3, Aggiunta giocatori

A questo punto inizia la partita vera e propria. Nella quarta scena (Fig.64), la visuale si sposta sul tavolo da gioco e tutti i partecipanti lanciano i propri dadi. Tra questi, uno viene scelto casualmente per “aprire le danze” e iniziare il round.



Fig 64: Scena 4, Inizio Round

Non appena il primo giocatore decide la scommessa (Fig.65), a seguire in senso antiorario, tutti gli altri giocatori aumentano la puntata, finché qualcuno non dubita.



Fig 65: Scena 5, Fase Scommesse

Vengono dunque posizionati tutti i bicchieri al centro e mostrati i dadi, rivelando chi ha vinto e chi ha perso il round (Fig.66). A quest'ultimo viene sottratto un dado e nel caso in cui non ne possieda più, viene eliminato dal gioco.



Fig 66: Scena 6, Dadi scoperti

Si riparte così dalla Scena 4, finché non termina l'intera partita.

L'ultima scena mostra in caso di vittoria, i festeggiamenti e la premiazione del personaggio scelto dall'utente (Fig.67), in caso di sconfitta la schermata di "Game Over".



Fig 67: Scena 7, Fine Partita

10.3 INTERAZIONE TRA FPGA E UNITY

Nella sottostante tabella viene schematizzato il funzionamento del protocollo di comunicazione realizzato per la trasmissione dei dati. Per ogni scena viene indicato il relativo stato interno della Control Unit, un esempio di possibili tasti premuti dall'utente e i segnali generati che attraversano il canale seriale.

SCENA	STATO	PULSANTI	CANALE SERIALE
<i>Schermata Iniziale</i>	IDLE		I (inizio partita)
<i>Selezione Personaggio</i>	AVATAR	 	D (personaggio successivo) D S (personaggio precedente) O (conferma personaggio)
<i>Aggiunta Giocatori</i>	NUM	 	D (aggiungi giocatore) D D S (rimuovi giocatore) A (conferma/ invio dati giocatori)
<i>Inizio Round</i>	INIT		3 (numero giocatori in campo) 0 (indice giocatore) 5 (numero dadi in possesso) 2 (primo dado) 3 (secondo dado) 6 (terzo dado) 1 (quarto dado) 2 (quinto dado) 1 (indice giocatore) 5 (numero dadi in possesso) .. T (turno giocatore) 2 (indice giocatore)
<i>Fase Scommesse</i>	INIT	 	B (scommette) 5 (dato scommesso) 2 (ricorrenza scommessa) T (turno giocatore) 0 (indice giocatore) D (incremento il dado) S (decremento il dado) O (confermo il dado scommesso: 1) O (conferma ricorrenza: x1) T 1 X (giocatore 1 dubita)
<i>Dadi Scoperti</i>	CHECK_WINNER		Z (scopri i dadi) 1(indice di chi ha perso)
...			
<i>Fine Partita</i>	FINISH		Y (fine partita)

Tabella 1: Interazione FPGA - Unity

11 CONCLUSIONI

11.1 RISULTATI CONSEGUITI

Tra le più grandi soddisfazioni raggiunte vi è sicuramente il fatto di esser riusciti a implementare un videogioco, solitamente sviluppato con tecnologie ad alto livello, sintetizzando un'architettura composta da reti logiche su una board FPGA.

Un traguardo altrettanto importante è quello di aver conseguito l'interfacciamento tra due sistemi eterogenei come la board FPGA e l'ambiente Unity. Al fine di raggiungere tale risultato è stato implementato il protocollo seriale da ambo le parti come illustrato nei paragrafi soprastanti.

Il sistema realizzato è inoltre altamente modulare, tant'è che nel caso in cui si volessero aggiungere funzionalità al gioco, non sarebbe necessario apportare grandi modifiche ai vari componenti. Qualora si volesse aumentare il numero massimo di giocatori, sarebbe infatti sufficiente cambiare il valore di una variabile. Se si volesse inoltre inserire una modalità multiplayer, non sarebbe necessario modificare alcun componente a livello di architettura.

Infine, un altro aspetto fondamentale del nostro progetto è costituito dall'implementazione dell'intelligenza del FPGA utilizzando principi appresi durante il corso di Fondamenti di Intelligenza Artificiale.

11.2 PROBLEMI RISCONTRATI

Nel corso dello sviluppo del progetto abbiamo dovuto affrontare diverse difficoltà, che ci hanno spesso obbligato a seguire strade alternative.

Uno dei problemi riscontrati fin dall'inizio è stata la generazione dei cinque dadi casuali durante l'inizializzazione di un giocatore. Infatti, sebbene avessimo la soluzione per la generazione di un numero casuale, questa non era adatta nel caso di cinque dadi, in quanto cinque numeri casuali assegnati durante lo stesso ciclo di clock avrebbero sempre lo stesso valore; questo ci ha dunque obbligati a temporizzare il processo.

Un'altra difficoltà è stata riscontrata durante l'implementazione del protocollo seriale per l'interfacciamento di Unity con la board FPGA.

Infine, la problematica riscontrata durante tutto lo sviluppo del progetto è stata il coordinamento nella comunicazione tra i componenti implementati in VHDL. Per questo motivo si è scelto di implementare un protocollo di Handshake durante lo scambio di messaggi per tutti i componenti, che desse conferma della corretta comunicazione.

11.3 SVILUPPI FUTURI E MIGLIORAMENTI

Grazie alla modularità del sistema sviluppato, è possibile integrare con facilità eventuali miglioramenti e funzionalità aggiuntive.

Una di esse potrebbe essere la possibilità di scelta, da parte dell'utente, delle varie regole da applicare, permettendo di giocare quindi tipologie di partite diverse tra loro.

Un'altra eventuale caratteristica da aggiungere potrebbe essere l'interfacciamento con un Joystick, invece che giocare utilizzando i tasti dell'FPGA.

Infine, un altro possibile sviluppo futuro considerato, potrebbe essere la modifica dell'euristica nell'AI_Controller. Si potrebbe ad esempio assegnare un fitness ad una scommessa calcolandolo attraverso la distribuzione di probabilità binomiale, introducendo quindi un nuovo componente per il calcolo della probabilità.

12 RIFERIMENTI

[1] Perudo:

<https://it.wikipedia.org/wiki/Perudo>

[2] FPGA:

https://it.wikipedia.org/wiki/Field_Programmable_Gate_Array

[3] Unity:

<https://unity.com/>

[4] Altera Quartus II:

<https://www.intel.com/content/www/us/en/programmable/downloads/software/quartus-ii-we/121.html>

[5] ModelSim:

<https://www.intel.it/content/www/it/it/software/programmable/quartus-prime/model-sim.html>

[6] RS-232:

https://it.wikipedia.org/wiki/EIA_RS-232