Implementazione della logica di gioco del Tris attraverso un FPGA

Samuele Ceccarelli, *374374*
Filippo Ottaviani, *353609*

***Abstract*—L'intenzione del progetto è quella di realizzare una stazione per il gioco del Tris attraverso la programmazione in VHDL della logica di un FPGA. Si vuole permettere ai due giocatori di immettere le proprie mosse tramite una pulsantiera e visualizzare lo stato attuale ed il risultato della partita sullo schermo del PC. Questo ha richiesto la creazione di una struttura gerarchica in VHDL, l'istanziamento di una connessione UART tra la scheda e il PC e l'implementazione di un'interfaccia grafica tramite Python.**

I. Introduzione

I cabinati arcade dei giochi più popolari hanno rappresentato una parte dell'infanzia di molti e sono riconosciute come una delle applicazioni ludiche più celebri dei sistemi embedded. A causa del tipo di utilizzo e delle specifiche da rispettare si sono preferiti sistemi dedicati a dispositivi *general purpouse*. Solo con l'avvento di nuovi giochi con requisiti più stringenti si è giunti alla necessità di piattaforme come le *console* che sono, a tutti gli effetti, dei computer prestati all'ambito videoludico. Questo ci ha motivato ad affrontare la progettazione di un sistema embedded che permettesse di sfidarsi ad uno dei giochi più semplici e più celebri del mondo: il tris (o *Tick Tack Toe*).
Il progetto ha richiesto due moduli fondamentali: quello logico e quello grafico. Il modulo logico si sarebbe dovuto occupare di leggere gli input dal tastierino elaborarli e scandire le diverse fasi di turni e controlli del risultato, inoltre avrebbe contenuto una sottosezione dedicata alla codifica dello stato della partita da restituire in uscita. Il modulo grafico, invece, avrebbe avuto il compito di interpretare lo stato della partita e mostrarlo all'utente in attesa della mossa successiva.
L'idea iniziale era quella di sfruttare l'uscita VGA della scheda FPGA per permettere all'utente di visualizzare la griglia, i turni ed eventuali notifiche di vittorie o pareggi su uno schermo, in modo da simulare in tutto e per tutto la struttura di cabinato classico. Come verrà argomentato in seguito, le difficoltà riscontrate nell'utilizzo di questa soluzione ci ha portato ad adottare una soluzione intermedia: la trasmissione dello stato della partita tramite UART dall'FPGA, responsabile del lato logico, ad un PC con uno script Python dedicato, "motore" del modulo grafico. Si è trattato del primo approccio a Python e a VHDL per entrambi e questo ci ha stimolati a dividerci equamente compiti e responsabilità.
La nutrita comunità di progettisti VHDL non ha, purtroppo, prodotto materiale completo e funzionante per quanto riguarda la logica di gioco del tris. Per quanto riguarda la comunicazione tramite UART, la documentazione in rete è copiosa e dettagliata; Vivado mette a disposizione gratuitamente per i possessori di licenza a scopo didattico degli IP appositi per questo tipo di interfaccia, si è scelto però di procedere con un codice dedicato più adatto alle nostre esigenze. Per quanto riguarda l'FPGA, invece, ci siamo basati sul suo datasheet, in particolare per la caratterizzazione degli ingressi Pmod e dell'uscita seriale. L'interfaccia grafica in linguaggio Python è stata realizzata grazie alle numerose guide in rete per l'utilizzo della libreria *Tkinter*, in particolare, ai dettagliati tutorial su *Youtube*.
UART è un protocollo di comunicazione seriale asincrono introdotto intorno agli anni 60' diventato presto uno standard per la comunicazione tra computer e dispositivi esterni. Viene tuttora utilizzato per lo scambio di informazioni nei microcontrollori e microprocessori grazie alla sua semplicità. Per il nostro progetto l'idea è quella di implementare la comunicazione UART tra l'FPGA della Basys 3 e il computer dove è in esecuzione uno script Python. Abbiamo tralasciato, il modulo di ricezione visto che ci interessava solo inviare comunicazioni codificate dall'FPGA e interpretaate dal PC. Questo perché è sufficiente inviare lo stato del gioco, in base ai bottoni premuti, senza l'attesa di un riscontro. L'implementazione *full-duplex* di questo protocollo in VHDL può risultare piuttosto complicata e poco intuitiva. Una possibile alternativa a questo tipo di scelta, è quella di implementare un IP core gratuito, fornito da Xilinx e disponibile direttamente nel software Vivado, che però, richiederebbe l'utilizzo di un microprocessore, come il Microblaze, e del linguaggio C con il software Vitis. Il protocollo UART si può concettualmente descrivere tramite la seguente figura:

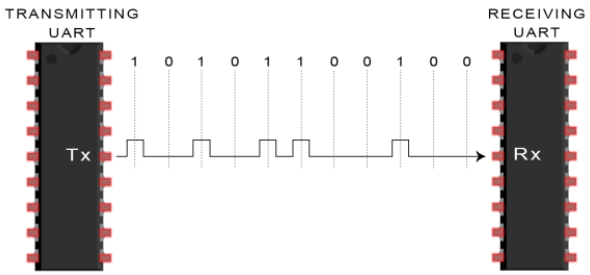

Fig. 1. Comunicazione UART

Quindi, l'implementazione della comunicazione via UART è stata realizzata prendendo spunto da degli esempi disponibili sulla piattaforma GitHub, rimodellando il codice e adattandolo alle nostre esigenze. Il resto dell'articolo è organizzato come segue: i lavori relativi al tema del progetto sono riportati nella

Sezione II. Nella Sezione III vengono invece descritte le fasi del progetto. Nella Sezione IV si riportano i risultati ottenuti. Le conclusioni finali si trovano, infine, nella Sezione V.

II. Lavori correlati

Le diverse ricerche in internet non ci hanno condotto ad articoli di particolare interesse ed aderenza alla nostra idea del gioco. Tuttavia, ci siamo soffermati su alcune pubblicazioni specifiche sulle tecnologie implementative e le abbiamo revisionate, in particolare abbiamo studiato quelli riguardanti l'interfaccia UART. Nel suo lavoro, U. Nanda [1], spiega e dimostra come i dispositivi FPGA sono una soluzione ideale per applicazioni di ricerca militare e universitaria. Inoltre, fa comprendere bene come l'architettura programmabile di un FPGA si offre molto bene allinterfaccia UART. L'implementazione è stata eseguita su un dispositivo Virtex-II, seguita da test hardware e verifica delle funzionalità dell'UART. Inoltre, viene spiegato in modo dettagliato il funzionamento del protocollo UART, per applicazioni anche più complesse che richiedono un controllo maggiore sui dati. Ci è comunque servito per avere un punto di vista aggiuntivo sul protocollo in questione. Un altro lavoro molto interessante e più vicino ai nostri interessi, è stato scritto da Y. Fang [2], che inizialmente descrive in modo semplice e comprensivo il funzionamento generico della comunicazione UART, utilizzando diagrammi e formule in modo diretto. Sono state poi eseguite delle simulazioni che verificano il funzionamento del sistema. Questo articolo è stato particolarmente importante poiché sono stati utilizzati parametri, e.g *baud rate*, con stesso valore di alcuni dei parametri del nostro lavoro. Un ultimo articolo, scritto da G. B. Wakhle [3], ci è servito per avere un punto di vista più completo. L'autore utilizza il linguaggio Verilog e conclude affermando che l'utilizzo di una FIFO (First-In-First-Out) porterebbe ad un netto miglioramento di flessibilità e stabilità, raggiungendo un numero maggior di bps (bit per secondo). In particolare, l'autore descrive molto bene il funzionamento della comunicazione UART in modo molto diretto e la rappresenta tramite una macchina a stati, dove ogni bit trasmesso/ricevuto rappresenta uno stato della macchina. Un altro lavoro importante che ci ha aiutato a realizzare la comunicazione UART, è stato quello di Scott Larson [4], che rende disponibile il suo codice in due versioni, una base e una migliorata. Il modulo in questione si occupa di eseguire il debouncing di un segnale in ingresso, restituendo in uscita il segnale stabilizzato.

[1] U. Nanda and S. K. Pattnaik, "Universal Asynchronous Receiver and Transmitter (UART)", 2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 2016, pp. 1-5, doi: 10.1109/ICACCS.2016.7586376.

[2] Y. -y. Fang and X. -j. Chen, "Design and Simulation of UART Serial Communication Module Based on VHDL," 2011 3rd International Workshop on Intelligent Systems and Applications, Wuhan, China, 2011, pp. 1-4, doi: 10.1109/ISA.2011.5873448.

[3] G. B. Wakhle, I. Aggarwal and S. Gaba, "Synthesis and Implementation of UART Using VHDL Codes," 2012 International Symposium on Computer, Consumer and Control, Taichung, Taiwan, 2012, pp. 1-3, doi: 10.1109/IS3C.2012.10.

[4] S. Larson, "Debounce Logic Circuit (VHDL)" https://forum.digikey.com/t/debounce-logic-circuit-vhdl/12573

III. Descrizione delle fasi del progetto

Il progetto è stato articolato in diverse fasi e in differenti flussi così da poter dividere il lavoro correttamente e avere chiari e scanditi i passi da compiere verso la realizzazione del sistema:

1) Caratterizzazione e studio del problema.
2) Analisi degli studi preesistenti e delle tecniche collaudate.
3) Sintesi di uno schema a blocchi logico che rappresentasse le fasi e le procedure del gioco.
4) Traduzione delle fasi del gioco in blocchi funzionali implementabili in VHDL.
5) Stesura dei codici delle relative entity.
6) Prova con testbench delle diverse componenti, prima separatamente, poi assemblando e testando i vari sottoblocchi del sistema complessivo.
7) Composizione delle grafiche e delle animazioni di gioco grazie a Photoshop.
8) Realizzazione del lato grafico tramite codice Python tramite l'approfondimento della libreria dedicata.
9) Prove relative alle animazioni e all'alternanza delle immagini a schermo.
10) Studio dell'interfaccia UART e valutazione delle alternative "off the shelf".
11) Implementazione del codice Python e della entity VHDL utile alla comunicazione seriale.
12) Collaudo del sistema finito.

A. Definizione del problema

Il primo approccio al problema è consistito nella stesura di un diagramma che descrivesse le procedure del Tris scegliendo una progettazione inizialmente astratta rispetto ad hardware e software per inquadrare correttamente le meccaniche semplici ma rigide del gioco.

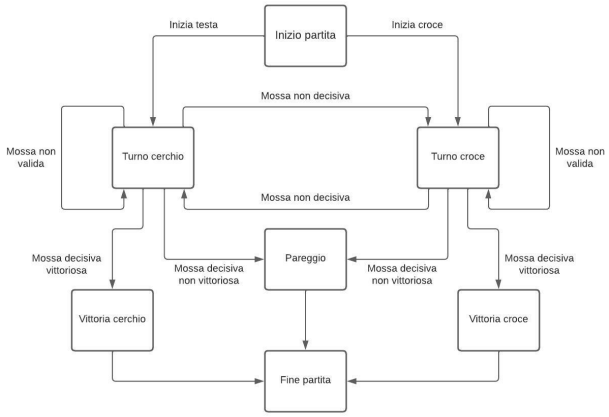

Fig. 2. Diagramma di una partita

La partita inizia con la griglia vuota 3x3 ed uno dei due giocatori che deve compiere la propria mossa scegliendo la casella dove apporre il proprio simbolo. Lo scopo è quello di creare una sequenza vincente verticale, orizzontale o obliqua di tre simboli. Ogni mossa può avere quattro esiti:

- Può essere una mossa vincente e quindi la partita finisce.
- Può portare ad un pareggio e anche in questo caso il gioco termina.
- Può essere una mossa non decisiva e quindi la partita continua cambiando d turno.
- Può essere scelta una casella già occupata e quindi non accade nulla, il giocatore deve cambiare selezione.

Le nove caselle della griglia verranno indicate per comodità con i valori tra 0 e 8 e ognuna di esse può essere libera, occupata da X oppure occupata da O. Il turno può essere di X o di O il risultato dopo ogni turno può corrispondere ad un pareggio, ad una vittoria di X, ad una vittoria di O oppure ad un nulla di fatto.

B. Traduzione nel diagramma di flusso

A partire dalla definizione precedente si è delineato uno schema algoritmico da implementare per poter gestire correttamente ogni mossa e ogni controllo del gioco. Segue, dunque, il diagramma appena descritto:

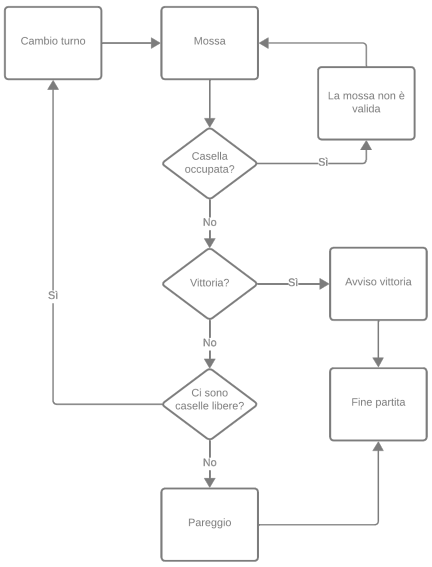

Fig. 3. Diagramma di flusso di un turno

Si giunge, dunque, alla conclusione che il sistema di gioco deve compiere tre controlli:

- Controllo dell'occupazione della casella selezionata.
- Controllo della vittoria in cui si scansiona la griglia di gioco per cercare una combinazione vincente di uno dei due giocatori.
- Controllo delle caselle libere per attestare un eventuale pareggio che si verifica quando non ci sono più caselle da occupare e nessuno ha vinto.

Da qui si ricavano le responsabilità del motore di gioco e l'ordine da imporre alle varie procedure. Inoltre, emerge un quadro chiaro su cosa occorre mostrare al giocatore a seguito del suo turno:

- La griglia aggiornata.
- Il turno successivo se l'ultima mossa non è stata decisiva.
- Il risultato se, invece, l'ultima mossa ha determinato una vittoria o un pareggio.

C. Individuazione dei blocchi di codice

Questo passaggio intermedio ci ha aiutato ad assegnare i giusti compiti ai differenti blocchi e, quindi, a raccogliere le procedure in moduli che comunicassero tra loro tramite segnali descrittivi dei singoli aspetti della partita. Ogni entità all'interno del sistema avrebbe dovuto elaborare questi segnali e produrre la propria uscita codificata. Nella prima fase si è deciso di assegnare ad ogni blocco i compiti descritti qui sotto:

- *inputReader:* legge l'input dei bottoni e codifica un comando da inviare a *turnManager* o direttamente all'uscita (*outEncoder*) nel caso di "accensione" o "spegnimento".
- *turnManager:* ogni volta che arriva un comando il blocco controlla se la casella è occupata e, se è libera, la occupa.
- *winChecker:* ad ogni aggiornamento della griglia (*gridX* o *gridO*) controlla se uno dei due giocatori ha vinto, se c'è stato un pareggio o se la partita deve continuare, restituisce quindi *result* che contiene l'esito di questi controlli.
- *outEncoder:* ad ogni aggiornamento dei segnali *gridX*, *gridO*, *turn*, *result* e *state* codifica questi nel uscita *outComm* e la invia al comparto grafico che si occuperà di serializzarla.

Per descrivere in maniera sintetica e comprensibile la comunicazione tra i blocchi si sono codificati i seguenti segnali che descrivessero lo stato della partita:

- *command:* rappresenta la casella selezionata dal giocatore dove apporre il proprio simbolo; assume valori da 0 a 8.
- *turn:* identifica a chi spetta la mossa: $'0'$ per O e $'1'$ per X.
- *move:* notifica che è avvenuta la mossa per poter invertire correttamente il turno; assume valori tra $'0'$ e $'1'$.
- *gridX:* segnala la presenza o meno di X su ogni casella. Consiste in un array di 9 elementi che assumono valori tra $'0'$ (quando assente) e $'1'$ (quando presente).
- *gridO:* segnala la presenza o meno di O su ogni casella. Consiste in un array di 9 elementi che assumono valori tra $'0'$ (quando assente) e $'1'$ (quando presente).
- *result:* descrive lo stato della partita attraverso due bit tramite la seguente codifica: "00" quando la partita è ancora in corso e nessuno ha vinto, "01" quando si verifica la vittoria di O, "10" quando si verifica la vittoria di X, mentre "11" in caso di pareggio.
- *outComm:* rappresenta l'uscita codificata a 24 bit del motore di gioco, consiste nella concatenazione dei segnali utili ad aggiornare il lato grafico: il primo bit, di controllo, è $'0'$ quando il comando deve aggiornare la partita e $'1'$ quando c'è uno spegnimento o un accensione (in questo caso tutti i bit successivi sono $'0'$). Nel caso di un uscita

legata alla partita in corso, dopo il primo bit seguono: *gridX*, *gridO*, *turn*, *result* e due bit impostati a "00" denominati *state*.

Di seguito si riporta uno schema riassuntivo della composizione del segnale a 24 bit *outComm* in uscita dal modulo logico del gioco:

Fig. 4. Codifica di outComm

Il vertice della gerarchia è rappresentato dall'entità *tris* che raccoglie tutti i component e i segnali discussi sopra per coordinare il sistema nella sua interezza. Dato, però che si tratti di entity composte da un process ciascuna si è preferito, una volta testate approfonditamente una ad una, racchiuderle in un unica entity chiamata, appunto, *tris*. Arrivati a questo punto il codice era suddiviso nelle tre entity: *tris*, *top* e *uart*.

Dopo un'attenta cernita delle possibili suddivisioni di responsabilità è stato prodotto il seguente schematico che illustra i processi appena descritti e i segnali usati per comunicare all'interno della entity *tris*.

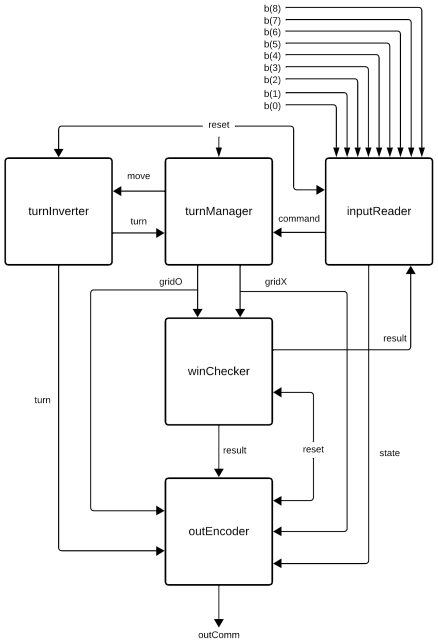

Fig. 5. Schema a blocchi del codice

D. Implementazione della macchina a stati

Riscontrate diverse simulazioni post-sintesi fallimentari, si è fatto un passo indietro alle simulazioni comportamentali tramite testbench; il fatto che queste ultime dessero risultati incoraggianti ci ha spinti ad indagare sulle possibili cause di questa asimmetria. Lunghe ricerche e numerosi tentativi ci hanno permesso di giungere alla conclusione che fosse necessario ridurre drasticamente l'utilizzo di clausole *if*, convertire le variabili in segnali e adottare, per quanto riguarda le meccaniche del turno, una macchina a stati. Le cause principali che ci hanno condotto a questo cambio di direzione sono state le inferenze dei latch relativi alle variabili che, in fase di sintesi, conducono a comportamenti imprevedibili e indesiderati. I latch possono essere difficili da identificare e possono essere nascosti all'interno di moduli complessi. Inoltre si è osservato che rendendo il segnale *command* sincrono rispetto al clock si evitavano anomalie nelle meccaniche di gioco quali: cambi repentini e indesiderati del turno, nuove segnature non richieste sulle griglie e mancate letture dei pulsanti. La macchina a stati realizzata è una macchina di Mealy asincrona che viene stimolata da *command*, un segnale sincrono rispetto al clock. Ad ogni esecuzione restituisce l'uscita *outC* e il segnale *next_state*, utile ad indicare lo stato in cui entrare nell'iterazione successiva.

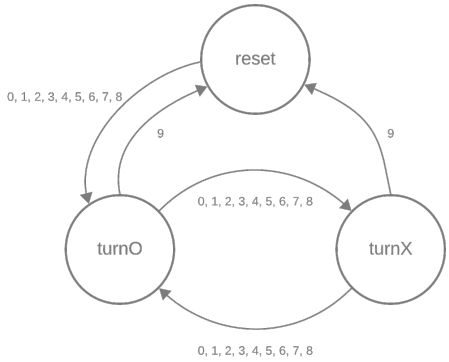

Fig. 6. Macchina a stati del turno

Il principio alla base della macchina a stati è l'alternanza dei due stati *turnX* e *turnO* che dato un comando (tra 0 e 8) segnano, come occupato, il bit corrispondente rispettivamente sulla griglia di X o su quella di O, verificano se uno dei due giocatori ha vinto e codificano l'uscita in base alla griglia complessiva, al turno ed al risultato. Se uno dei due giocatori preme il tasto reset lo stato attivo è *reset_state* che svuota le griglie, riporta il risultato al valore "00" e assegna il turno ad O.

E. Realizzazione delle grafiche

Sin dall'inizio il lato grafico ha avuto, per noi, un peso importante seppure non fosse il centro della trattazione. Come già detto, in prima istanza si era organizzato il progetto per sfruttare l'uscita VGA della scheda ma l'elevata complessità delle soluzioni trovate online ci ha suggerito di virare verso direzioni compatibili con i tempi a disposizione. In particolare l'uscita VGA richiedeva una gestione articolata della memoria e delle immagini difficilmente affrontabile senza l'implementazione di un microprocessore e di tutto ciò che esso comporta. Anche grazie alla segnalazione della professoressa Placidi e di Riccardo Nuti della ART SpA abbiamo scelto di trattare l'uscita seriale UART di cui abbiamo discusso

in precedenza. Le immagini create per l'occasioni sono le seguenti:

- La griglia vuota.
- Il simbolo X.
- Il simbolo O.
- Vittoria di X.
- Vittoria di O.
- Pareggio.
- Turno X.
- Turno O.

Inoltre si sono voluto realizzare due animazioni per l'accensione e per lo spegnimento del sistema; essendo gestite dal lato grafico non vanno a disattivare il sistema di gioco su FPGA, ma solo il lato grafico supportato da Python. Si riporta ora la posizione sullo schermo delle caselle della griglia dove andranno posizionati simboli X e O e la porzione dedicata ai turni e i risultati:

Immagine	X di origine	Y di origine
Casella 0	170	90
Casella 1	274	90
Casella 2	375	90
Casella 3	170	194
Casella 4	274	194
Casella 5	375	194
Casella 6	170	294
Casella 7	274	294
Casella 8	375	294
Turni/risultati	212	410

Per quanto riguarda la dimensione delle immagini la tabella riassuntiva è la seguente:

Immagine	Lunghezza	Larghezza
Griglia	640	480
Animazione d'ingresso	640	480
Animazione d'uscita	640	480
Turno	214	41
Risultato	214	41
Simbolo	92	92

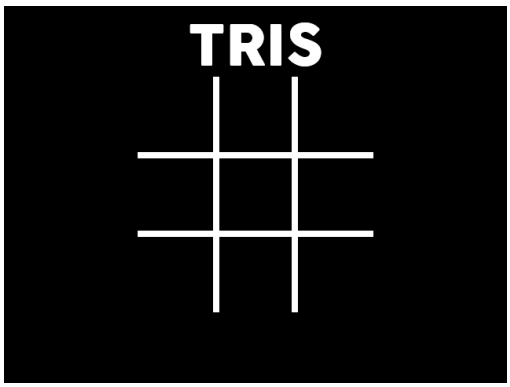

Fig. 7. Griglia vuota

Fig. 8. Turno di X

F. Grafica in Python

Da numerose ricerche in rete abbiamo appreso che lo strumento più versatile e diffuso per l'animazione di componenti grafiche a schermo in Python è *Tkinter*, una libreria basata su toolkit gratuiti come *Tcl* e *Tk*. Le sue funzioni principali sono:

- Creazione di finestre e widget.
- Gestione di eventi.
- Grafica e animazione.

In particolare abbiamo scelto di usare *PhotoImage*, una classe che consente di caricare e visualizzare immagini nelle proprie applicazioni. PhotoImage può gestire una varietà di formati di immagini, inclusi GIF, PNG, PGM e PPM. Ciò significa che è possibile utilizzarla per visualizzare immagini da un'ampia gamma di origini, inclusi i propri file e risorse online. Per utilizzare PhotoImage, è necessario prima creare un oggetto PhotoImage chiamando il costruttore associato. Il costruttore accetta un numero di parametri, tra cui il percorso del file immagine o i dati dell'immagine stessa. Una volta creato un oggetto PhotoImage, è possibile assegnarlo a un widget Tkinter, come un Label o un Button. Ciò visualizzerà l'immagine sul widget.
Il codice Python è stato strutturato in tre script:

- *graphicEngine:* istanzia la finestra, carica tutte le immagini utili e definisce tutte le funzioni utili:
  - *init():* mostra la griglia vuota e il turno del giocatore che inizia.
  - *placeX(cell):* posiziona una X nella cella indicata.
  - *placeO(cell):* posiziona una O nella cella indicata.
  - *turnChange(turn):* mostra il turno in corso.
  - *endGame(result):* mostra il risultato della partita
  - *refresh():* svuota la finestra
  - *animationStart():* fa partire l'animazione d'entrata.
  - *animationFinish():* fa partire l'animazione d'uscita.
- *commandReader:* decodifica il comando arrivato dalla UART, richiama la funzione corrispondente e passa il parametro associato.
- *main:* legge i dati in arrivo dalla seriale e ricompone il commando.

Una volta finito di scrivere il codice Python si sono svolte diverse prove tramite comandi di test e una sequenza temporizzata di funzioni richiamate per verificarne la correttezza.

G. Implementazione UART

Per la comunicazione seriale, si è scelto di scrivere un codice semplificato rispetto a quelli reperibili online. La trasmissione di 1 byte, ad esempio, prevede 11 bit totali: il primo è lo *start bit* che indica l'inizio della transazione, seguito da 8 bit di dati, un bit per la parità (opzionale) e, infine, uno di stop. Nel nostro caso, si è deciso di non utilizzare il bit di parità per il controllo degli errori vista la semplicità di utilizzo del protocollo. Inoltre, poiché il protocollo lavora ad una frequenza minore rispetto a quella del clock principale fornito dalla Basys 3, è bene utilizzare delle tecniche di divisione del clock, in modo da avere il parametro *baud rate* corretto. Quest'ultimo, esprime la velocità con cui un numero di bit viene inviato/ricevuto tramite UART. Nel nostro caso, poiché

il clock fornito dalla scheda è pari a $100MHz$ e il *baud rate* è 9600, si è scelto di implementare un semplice divisore di clock tramite un contatore. Il numero di cicli di clock che deve aspettare il trasmettitore prima di inviare i dati è pari a:

$$N_{cicli} = \frac{f_{clock}}{baudrate} = \frac{100000000}{9600} = 10416$$

H. Realizzazione pulsantiera

Per permettere agli utenti di immettere la propria scelta durante il turno è stata realizzata una pulsantiera con 9 tasti che rappresentassero le nove caselle del gioco del tris. Sono serviti bottoni, jumper, breadboard, resistori da 10 kΩ e cavi, componenti facilmente reperibili in un qualunque kit base per Arduino. L'intenzione iniziale era quella di creare una griglia di pulsanti per fornire un'interfaccia più coerente, le dimensioni della breadboard, però, hanno permesso soltanto una configurazione in linea. Segue una rappresentazione schematica di quanto appena descritto:

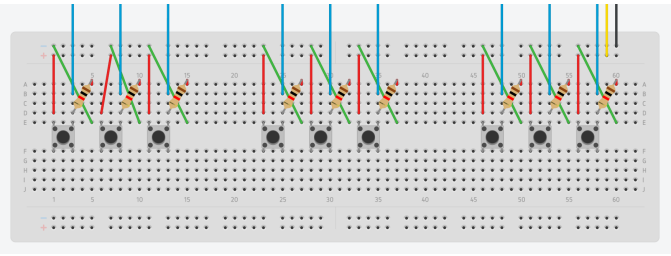

Fig. 9. Pulsantiera

Una volta assemblata la pulsantiera sono stati collegati i fili di segnale e di alimentazione alla porta Pmod seguendo lo schema fornito nel datasheet della Basys 3.

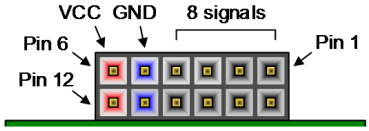

Fig. 10. Ruoli dei pin Pmod

Dopo aver realizzato il collegamento è stato testato grazie ad un codice VHDL per accendere i led sulla scheda a seconda del pulsante premuto in modo da verificare la correttezza dei collegamenti. Il reset è stato assegnato ad uno dei pulsanti presenti sulla scheda Basys 3.

I. Debouncing dei pulsanti

L'uso di un resistore da 10 kΩ in serie ad ogni pulsante è fondamentale per mantenere stabile il loro segnale logico, poiché questi possono essere affetti da bouncing. Il *bouncing* o rimbalzo è un fenomeno meccanico che si verifica quando viene premuto o rilasciato il pulsante, esattamente quando il contatto di metallo all'interno del pulsante chiude o apre il circuito. Durante questi eventi, il contatto meccanico può oscillare molto rapidamente prima di stabilizzarsi, poiché la velocità di passaggio della corrente è molto maggiore dell'azione meccanica. Questo può causare, quindi, falsi segnali, valori casuali o jitter nel circuito. In alcuni casi, si può anche utilizzare un condensatore in parallelo, che si carica o si scarica gradualmente attraverso la resistenza del pulsante, rallentando la variazione del segnale e riducendo l'effetto dei rimbalzi. Nel nostro caso, però, è stato sufficiente l'utilizzo di soli resistori. È risultato, quindi, necessario implementare una logica di debouncing per ogni pulsante utilizzando un codice reperibile dal sito DigiKey.com. Abbiamo aggiunto al codice discusso finora i 9 component che si occupano di eliminare questo fenomeno dalla pulsantiera. Il codice è stato sviluppato da Scott Larson, e come riportato dallo stesso, permette di eseguire il debouncing di un ingresso specificando, tramite un parametro attributo, l'intervallo di tempo nel quale il segnale deve mantenere un valore stabile. La scelta dell'utilizzo di questo codice è basata sulla sua semplicità strutturale, poiché esistono molte altre implementazioni, più o meno complesse, ideate per diverse esigenze. Abbiamo deciso di mantenere il parametro impostato a 10 ms, valore di default stabilito dal programmatore.

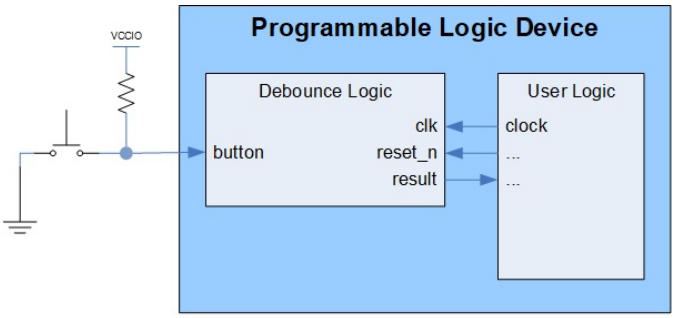

Fig. 11. Schema del debouncer

IV. Risultati sperimentali

A. Introduzione

Lo svolgimento del progetto ha imposto un discreto numero di prove e test intermedi grazie all'approccio modulare adottato e alla natura gerarchica del design VHDL da noi scelto. In particolare si è scelto di includere nella relazione dei risultati sperimentali anche quelli negativi riscontrati in assenza della macchina a stati.

B. Test della logica di gioco senza macchina a stati

L'argomento delle prime sperimentazioni è stato la lettura della pulsantiera tramite l'interfaccia Pmod dell'FPGA. A questo scopo è stato realizzato un sottoprogetto che permise di leggere gli ingressi del Pmod, che ospitavano i pin connessi ai bottoni e accendere dei LED precedentemente mappati 1-a-1 con l'array della pulsantiera. Una volta accertata la bontà della configurazione e degli ingressi Pmod, si è passati alla logica di gioco che, come detto, inizialmente prevedeva le 5 entity poi confluite nel *tris* e l'entity relativa alla UART oltre, ovviamente, al top della gerarchia. Per provare la parte della logica di gioco, è stato realizzato un testbench VHDL e sono state effettuate simulazioni di partite ripetute comprendendo le

varie combinazioni di vittorie e pareggi. Dopo aver sistemato il codice e corretto eventuali errori individuati si è iniziato a testare le interazione tra i moduli sopra descritti tramite testbench dedicati. In particolare si è proceduto con i testbench dei seguenti blocchi:

1) turnManager e turnInverter.
2) inputReader, turnManager e turnInverter.
3) winChecker, inputReader, turnManager e turnInverter.
4) outEncoder, winChecker, inputReader, turnManager e turnInverter.

Aggiungere un'entity alla volta alla gerarchia durante i test ha permesso di verificare le interconnessioni e il sincronismo tra loro.

Fig. 12. Testbench di tutta la logica di gioco

Il testbench mostrato consiste nella sequenza riportata qui sotto:

1) Accensione
2) Reset
3) Mossa di X in 0
4) Mossa di O in 1
5) Mossa di X in 3
6) Mossa di O in 4
7) Mossa di X in 6
8) Mossa di O in 7
9) Mossa di X in 8
10) Reset
11) Mossa di O in 7

La simulazione comportamentale ha confermato la correttezza delle dinamiche di cambio turno, controllo vittoria e aggiornamento griglia. Il reset ha sortito gli effetti desiderati e le griglie ad ogni passaggio sono risultate coerenti rispetto alla mossa appena conclusa. Nel testbench si sono specificate le mosse da compiere e gli intervalli da attendere grazie alle funzioni *wait for* che sospendono l'esecuzione per il tempo specificato.

La sintesi e la conseguente simulazione hanno, però, rivelato le criticità del lavoro svolto, spingendoci a rielaborare il codice nella sua interezza. Come riportato nella sezione dedicata alle fasi di realizzazione, si è ripensato il progetto nella direzione della macchina a stati.

C. Test della logica con la macchina a stati

Dopo gli insuccessi delle simulazioni post-sintesi, come argomentato in precedenza, si è scelto di ridurre le entity, le variabili e sincronizzare gli ingressi con il clock. Questo cambio di rotta è stato validato dalle simulazioni post-sintesi e post-implementazione che, finalmente, attestavano il funzionamento del nuovo sistema. I testbench e gli ingressi simulati sono i medesimi della sezione precedente in quanto l'implementazione della macchina a stati non ha imposto alcuna modifica ai segnali in ingresso ed in uscita.

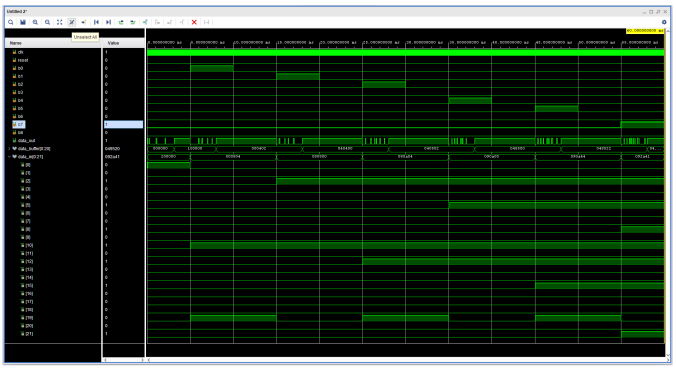

Fig. 13. Simulazione post-sintesi

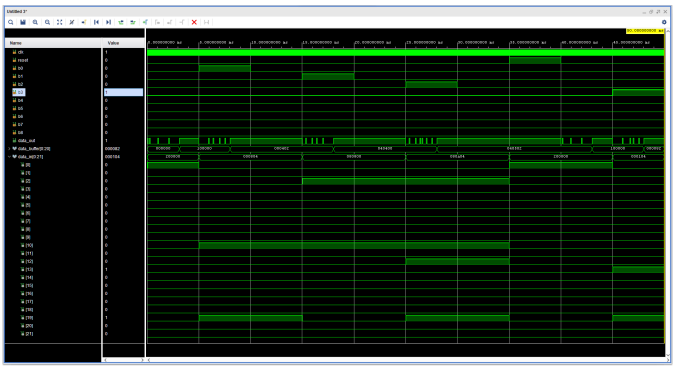

Fig. 14. Simulazione post-implementazione

Come previsto l'aggiunta dei componenti per il debouncing non hanno introdotto novità nelle simulazioni tranne il ritardo di 10 ms nella risposta del sistema alla pressione dei bottoni.

D. Test del codice Python

La fase di test successiva ha riguardato il comparto grafico del progetto ossia il codice Python e le immagini mostrate a schermo all'arrivo dei comandi della seriale. Per mantenere separati gli ambiti si è scelto di verificare innanzitutto la bontà delle animazioni e la correttezza delle posizioni dei singoli elementi della finestra. A questo scopo si sono richiamate le funzioni corrispondenti alle singole animazioni singolarmente e in sequenza separate da brevi intervalli di tempo. La sequenza finale per la prova corrisponde al seguente elenco di comandi:

1) Animazione di avvio
2) Reset
3) Cambio turno
4) O sceglie la casella 0
5) Cambio turno

6) X sceglie la casella 1
7) Cambio turno
8) O sceglie la casella 2
9) Cambio turno
10) X sceglie la casella 3
11) Cambio turno
12) O sceglie la casella 4
13) Cambio turno
14) X sceglie la casella 5
15) Cambio turno
16) O sceglie la casella 6 (O vince)
17) Animazione di chiusura

Come si può notare dalla sequenza, la grafica realizza solo il comando assegnato, non ha controllo sulle dinamiche di gioco, non rileva le vittorie e non conosce la nozione di turno se non per l'immagine ad esso associata. Il reset, implementato dalla funzione *refresh()*, ha il solo scopo di iniziare una nuova partita "pulendo" l'interfaccia.

E. Test della UART

Il passo successivo nelle sessioni di test è stato il collaudo dell'interfaccia UART che si è articolato in due passaggi:

1) Prova del meccanismo di serializzazione tramite testbench e simulazione.
2) Prova di implementazione sull'FPGA tramite input da bottoni.

Il primo test ha richiesto l'utilizzo di un testbench per verificare che la entity convertisse correttamente il segnale in ingresso (*std_logic_vector*) in una sequenza di uscite, ad un bit coerente. Il testbench inviava segnali intervallati in base al bottone premuto e la entity *UART.vhd* pilotava il pin *tx* di conseguenza, per poi rimanere in attesa di nuovi segnali mantenendo il livello logico alto, come richiesto dallo standard utilizzato. La simulazione è stata effettuata attraverso vettori di 8 bit (per facilitare la lettura) forzando il clock e due onde quadre separante nel tempo per i due bottoni; l'esito della simulazione è riportato di seguito.

Fig. 15. Simulazione dell'interfaccia UART

Come si può notare il primo pulsante porta all'uscita "01010101" mentre il secondo a "01110000" come previsto; il test è stato ripetuto con vettori a 24 bit e il risultato è stato altrettanto soddisfacente. Il clock è stato impostato con un periodo di 10 ns per poter apprezzare il funzionamento con la *baud rate* reale, 9600.

F. Test finale

Infine, le prove conclusive hanno coinvolto il sistema assemblato prevedendo partite con i vari esiti e scenari: avvio, nuova partita e spegnimento del sistema con le relative animazioni e conseguenze sullo stato della partita.

V. Conclusioni

L'esperienza ricavata dallo svolgimento di questo progetto risulta preziosa per entrambi nell'ottica di ottenere conoscenze trasversali negli ambiti hardware e software. Confrontare le nostre capacità con una sfida nuova ha arricchito il bagaglio di competenze di entrambi. La continua ricerca di soluzioni funzionali alle nostre idee ha, senza dubbio, ampliato i nostri orizzonti e ci ha invitato alla scoperta della disciplina del codesign.
Dal punto di vista del progetto, la scarsità di risorse in merito alla UART e ad implementazioni in VHDL del Tris ci ha stimolato a dare il nostro contributo. Una delle evoluzioni future del progetto potrebbe essere l'implementazione della VGA per visualizzare le immagini su uno schermo oppure un approfondimento sul collegamento UART provando ad aggiungere logiche di controllo del bit di parità e *acknowledgement*. Altri futuri interventi potrebbero riguardare la qualità delle animazioni e un miglioramento complessivo della interfaccia utente e dell'esperienza di gioco. Il codice prodotto verrà caricato su *GitHub* come progetto *open source* per accogliere contributi e suggerimenti di progettisti esperti e non.