SISTEMA DI COLLAUDO PER DECODIFICATORI IMPLEMENTATI SU FPGA

RELATORE: *Prof.* Daniele Vogrig

LAUREANDO: Daniele Caliolo

A. A. 2009-2010

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA

SISTEMA DI COLLAUDO PER DECODIFICATORI IMPLEMENTATI SU FPGA

RELATORE: *Prof.* Daniele Vogrig

LAUREANDO: Daniele Caliolo

Padova, 20 Novembre 2009

Alla mia Famiglia, ai miei Amici.

La teoria è quando si sa tutto, ma non funziona nulla.
La pratica è quando funziona tutto, ma non si sa perché.
Infine, si finisce per coniugare teoria e pratica:
non funziona nulla e non si sa perché.
- Albert Einstein

INDICE

Introduzione

Lo scopo di questa tesi è di realizzare un sistema di collaudo per un decodificatore basato sull'algoritmo di Viterbi. Si è cercato di mantenere un approccio quanto più modulare possibile, in modo da poter impiegare il sistema anche nella verifica di altri dispositivi apportando piccole modifiche, in particolare nell'applicazione di controllo, nel metodo di codifica dei dati.

Il sistema realizzato si compone essenzialmente di due parti: la prima, scritta in VHDL per l'implementazione su FPGA, descrive un'interfaccia verso il dispositivo in esame e comprende il microcontrollore che comunica con il PC tramite un dispositivo UART RS232 e le memorie FIFO per i dati da e verso il decoder; la seconda sezione è l'applicazione di controllo sviluppata in Java che genera i dati, simulando errori di comunicazione, e verifica la corretta elaborazione di questi.

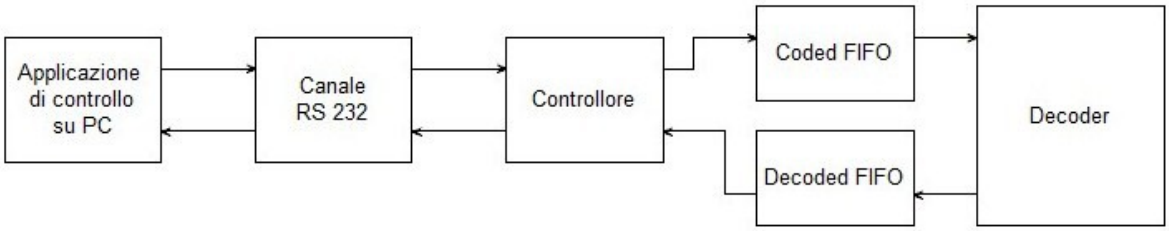

Figura 0.1 – Struttura del progetto

Nei successivi capitoli verrà illustrato l'hardware a disposizione, il canale di comunicazione impiegato, i dispositivi implementati sull'FPGA e il firmware scritto per il processore oltre a spiegare il codice Java dell'applicazione di controllo e mostrare il funzionamento del sistema di collaudo.

Capitolo 1

Hardware, Software e Interfaccia di Comunicazione

1.1 I dispositivi programmabili

I circuiti logici programmabili [1,2], meglio noti come *PLD (Programmable Logic Device)*, costituiscono un nutrito gruppo di componenti elettronici il cui comportamento viene definito tramite programmazione da parte del progettista anche successivamente al momento in cui il dispositivo viene montato nel circuito, al contrario dei normali circuiti integrati che vengono progettati e realizzati per uno specifico uso.

I primi componenti programmabili fecero la loro comparsa negli anni 70, basati sulla tecnologia ad antifusibile al silicio: un componente che normalmente si comporta come isolante ma che in seguito ad opportune sollecitazioni in tensione diventa conduttore, permettendo quindi di realizzare collegamenti tra elementi logici e quindi ottenendo semplici funzioni logiche combinatorie. Pregio di tale tecnologia è la non volatilità, per la quale sono ancora utilizzati nella realizzazione di PROM e di applicazioni speciali (aerospaziali, militari, biomedicali); d'altro canto gli alti costi, la difficoltà e lentezza di programmazione oltre alla non reversibilità sono caratteristiche che ne decretarono inevitabilmente la decadenza.

Con il diffondersi della tecnologia CMOS, verso la metà degli anni 80, fu possibile integrare un sempre maggior numero di componenti per unità di area permettendo quindi di realizzare circuiti sempre più complessi; si iniziò a collegare più blocchi PLD tramite una matrice di interconnessione dando alla luce i *CPLD (Complex PLD)* che ebbero successo grazie alla possibilità di riprogrammare i dispositivi impiegando memorie SRAM: e` così possibile associare ad ogni cella di memoria un transistor MOS, programmando quindi le connessioni come aperte o chiuse in funzione del valore memorizzato.

Venne anche sviluppata la tecnologia *PLA (Programmable Logic Array)*, basata essenzialmente su matrici di connessioni per realizzare funzioni AND e OR, ottenendo funzioni logiche a n ingressi definendo quali ingressi o quali risultati intermedi dovessero essere collegati a una specifica porta logica integrata nel componente (v. fig. 1.1).

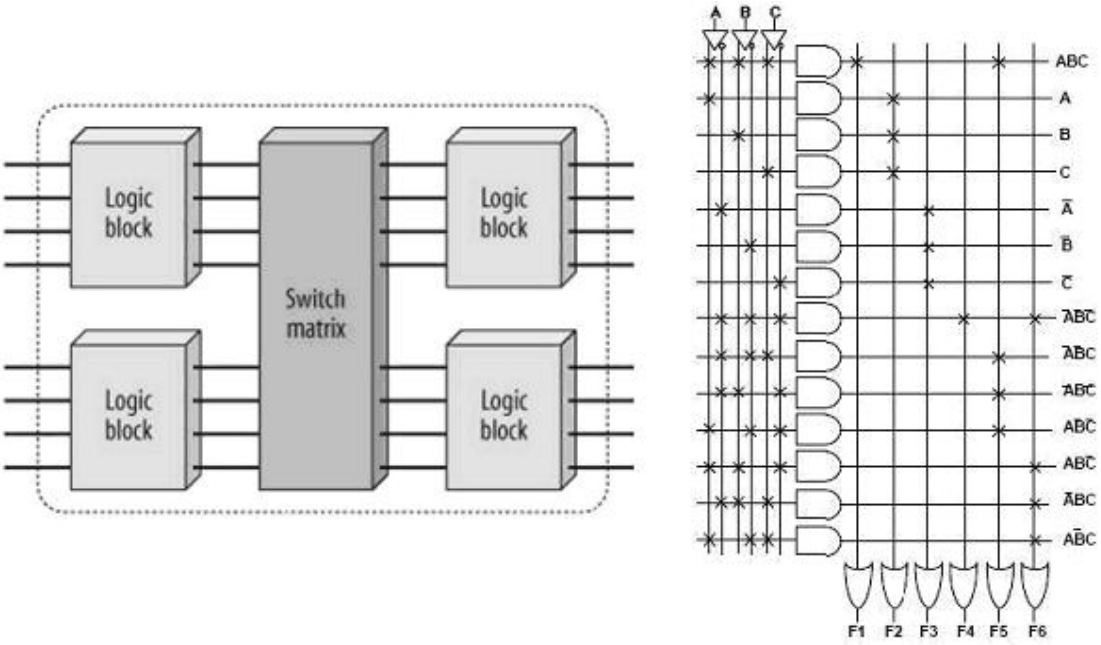

Fig. 1.1 ***Architetture CPLD e PLA***

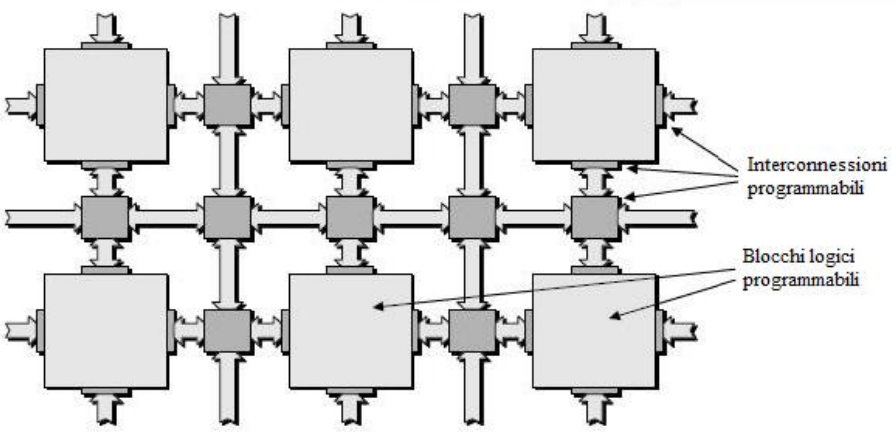

Figura 1.2 – Architettura FPGA

1.2 L'FPGA

Con lo svilupparsi della tecnologia, vennero realizzati CPLD sempre più complessi che permettevano una totale interconnettività tra i blocchi logici (Mega PAL), che furono però un fallimento.

La svolta si ebbe nel 1984, quando Xilinx presentò i primi FPGA (Field Programmable Gate Array) costituiti da un numero elevato di blocchi logici programmabili collegati da una maglia di connessione composta da bus incrociati e matrici di commutazione programmabili (v. fig. 1.2).

Rispetto a un circuito integrato tradizionale, un FPGA presenta caratteristiche molto interessanti: permette infatti di ridurre i tempi di sviluppo, di abbattere i costi di realizzazione per piccoli volumi (v. fig. 1.3) e, grazie alla riprogrammabilità, consente di effettuare rapidamente modifiche al circuito implementato. Anche la fase di prototipizzazione risulta facilitata grazie all'impiego di opportuni ambienti di sviluppo e linguaggi di programmazione come il VHDL o il Verilog, dai quali vengono ricavati i bitstream necessari alla programmazione dell'FPGA. D'altro canto, la necessaria generalità del componente porta a un non completo impiego delle risorse disponibili e non permette sempre una completa ottimizzazione del dispositivo realizzato.

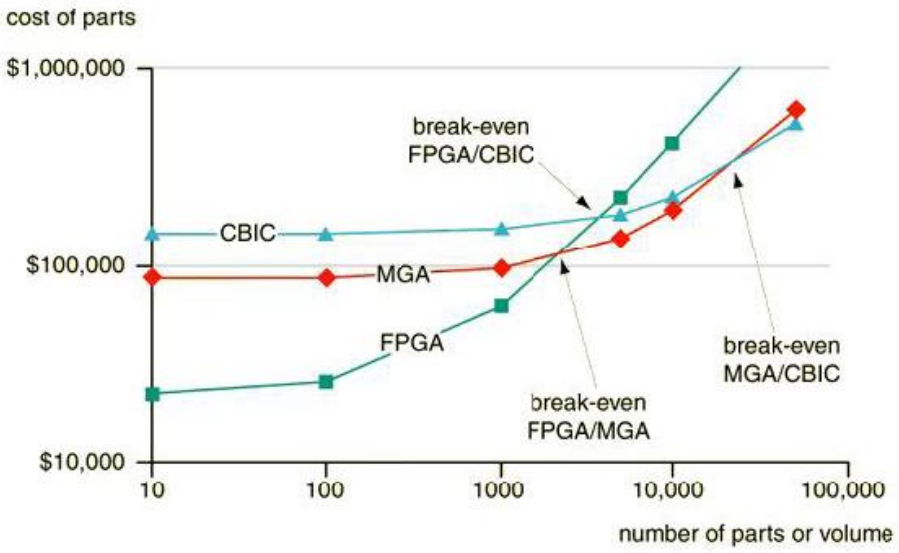

Fig. 1.3 - Costi di produzione per varie tecnologie

1.3 L'FPGA Xilinx Spartan 3 e la scheda S3SKB

L'FPGA utilizzata in questo progetto appartiene alla famiglia Spartan3 della Xilinx [3]. Come tutte gli FPGA è suddivisa in elementi fondamentali denominati CLB (*Configurable Logic Block)* i quali si interfacciano verso la matrice di interconnessione permettendo il collegamento di più CLB tra loro. A loro volta, i CLB sono costituiti da due coppie di slices asimmetriche: ad ogni coppia corrisponde un percorso di riporto per velocizzare le funzioni di calcolo e mentre una coppia di slices è dedicata esclusivamente alla realizzazione di funzioni logiche combinatorie, l'altra è utilizzabile anche come RAM distribuita o shift register. In figura 1.4 è riprodotta la struttura di una slices dove troviamo:

- 2 LUT (*LookUp Tables*) a 4 ingressi, per la definizione di funzioni logiche;
- 2 flip-flop per la realizzazione di RAM distribuita o di circuiti logici sincroni;

- Una catena di riporto per ottimizzare i circuiti matematici come sommatori, moltiplicatori, etc. (il segnale passa per soli tre multiplexer);
- Porte logiche dedicate per la realizzazione di circuiti matematici;
- Multiplexers programmabili e non, per realizzare funzioni logiche a più di 4 ingressi e per determinare il comportamento delle uscite.

Ogni CLB è inoltre connesso direttamente ai CLB confinanti tramite bus e contemporaneamente alla matrice di interconnessione per realizzare collegamenti tra CLB distanti (v. fig. 1.5).

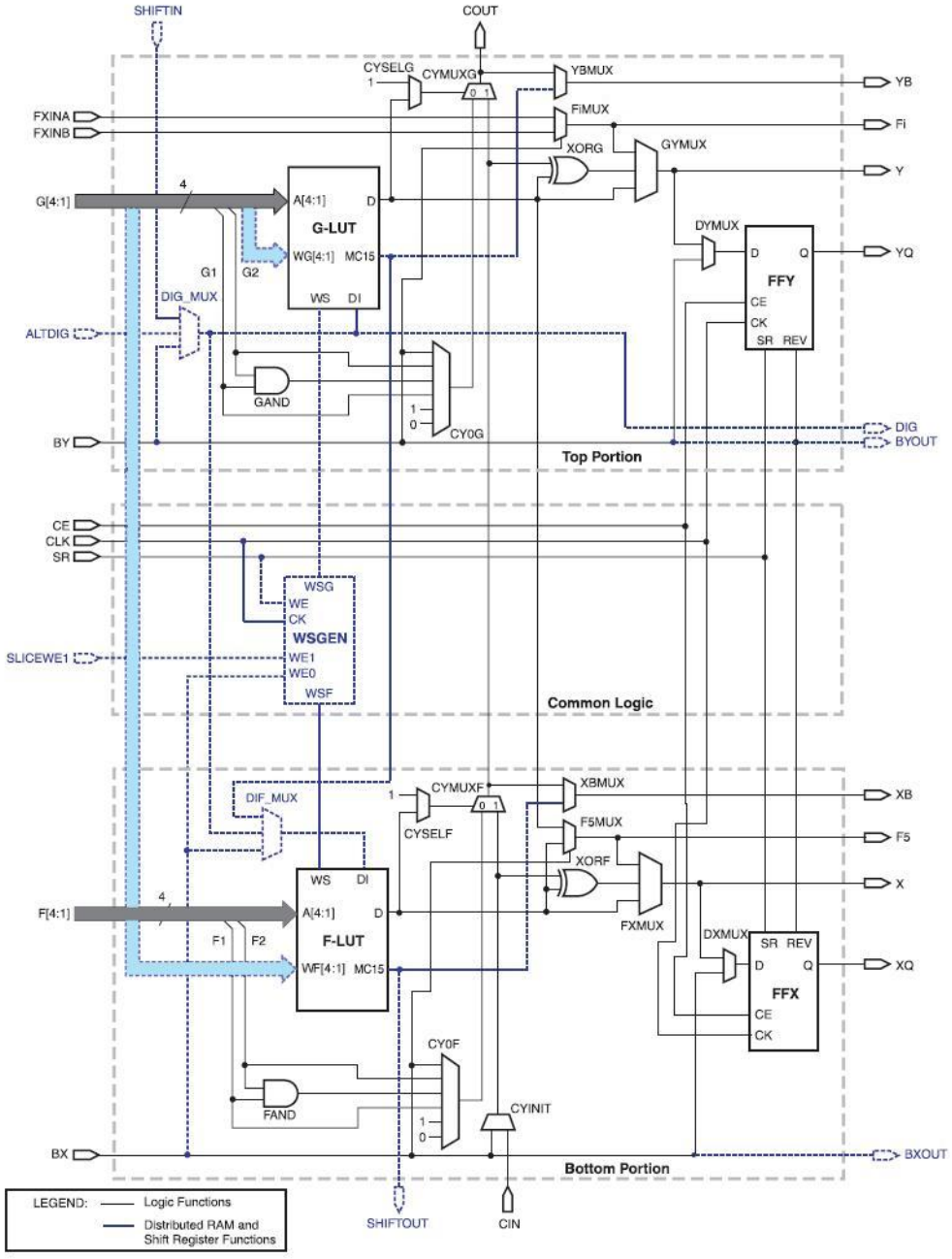

Figura 1.4 – Struttura di una slice

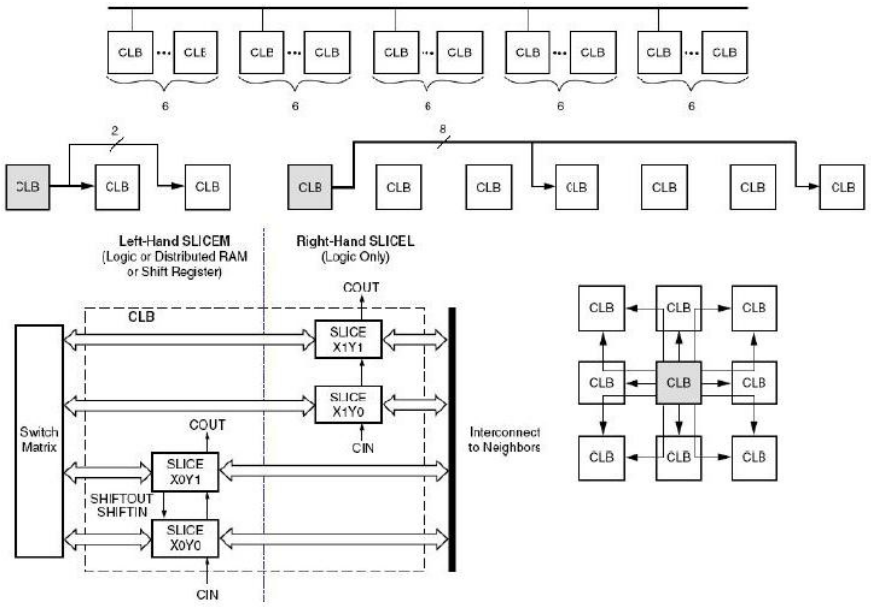

Fig. 1.5 - Struttura dei CLB e schema dei collegamenti

Il progetto realizzato si basa sull'uso della scheda Digilent S3SKB [4] (*Spartan3 Starter Kit Board*), visibile in fig. 1.6, sulla quale sono montati:

- FPGA Xilinx XC3S200 FT256: 200k gate, 12 block RAM per totali 216 kbit, 173 I/O;
- Memoria Flash Xilinx XCF02S;
- Due moduli memoria RAM Issi 256k x 16;
- Porta VGA 8 colori;
- Porta PS/2;
- Porta seriale RS-232 con traslatore di livello;
- Interfaccia JTAG per la programmazione;
- 4 pulsanti;
- 8 switch;
- 4 display LED 7 segmenti in multiplexing;
- 8 LED;
- 3 bus di espansione;
- Oscillatore al quarzo 50 MHz.

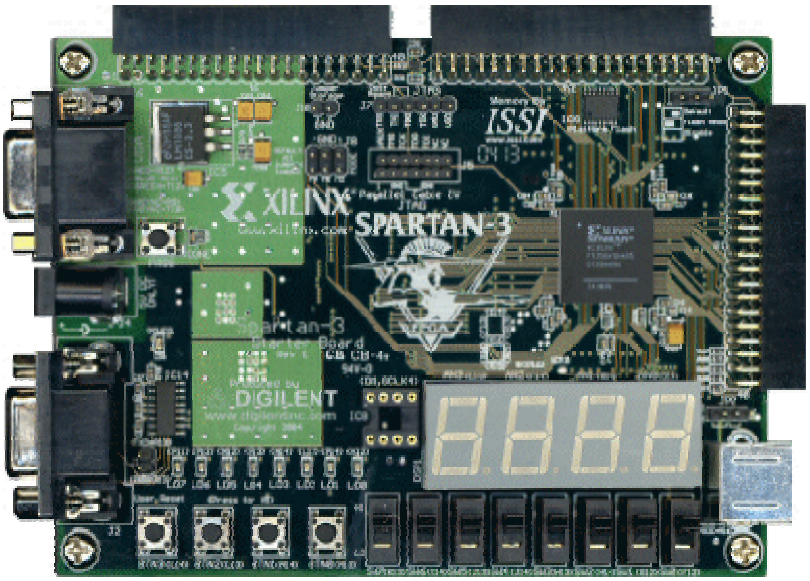

Fig. 1.6 - Scheda Digilent S3SKB

Tale scheda, è stata scelta per la buona completezza della dotazione oltre all'approfondita conoscenza di questa, acquisita durante il corso di Laboratorio di Elettronica Digitale.

Nel progetto, oltre ovviamente all'FPGA, verranno utilizzati i LED e i display per visualizzare lo stato del sistema (selezionando tramite gli switches il dato interessato), un pulsante come reset asincrono e la porta RS-232 per la supervisione del collaudo da parte del PC. Durante lo sviluppo dell'applicazione di supervisione, è risultato inoltre molto utile poter salvare le programmazioni di prova nella memoria flash permettendo quindi di riprogrammare in maniera agevole il sistema senza dover collegare il cavo JTAG e utilizzare il software di sviluppo ISE.

1.4 Il Java

Con il continuo aumento della richiesta di prodotti elettronici, i produttori hanno dovuto far fronte alla necessità di aumentare i volumi di produzione mantenendo comunque alta la qualità [5]. Nasce quindi l'esigenza di velocizzare e automatizzare quanto più possibile le procedure di controllo, cui può rispondere ampiamente l'informatica: sviluppando opportune applicazioni, un PC può gestire un numero elevato di test contemporanei, confrontando le misure con parametri di progetto e decidendo quindi autonomamente se il pezzo in esame sia da scartare o da approvare oltre a redigere report dettagliati sui risultati. Inoltre si esclude il fattore umano dai test riducendo drasticamente il numero di errori nei test, spostando quindi l'operatore dal banco di prova ai quadri di supervisione e controllo.

Nel progetto in esame, si è da subito denotata la necessità di realizzare una applicazione che permettesse di gestire un collaudo con consistenti volumi di dati, ma il dubbio principale era sul linguaggio di programmazione più appropriato.

Esistono al giorno d'oggi molti linguaggi estremamente potenti: dal C al MatLab, dal C++ al Java. La scelta è ricaduta su quest'ultimo principalmente per l'altissima portabilità del codice [6], la cui esecuzione è demandata alla cosiddetta Virtual Machine (JVM). Nel caso si fosse scelto un altro linguaggio, avremmo dovuto definire il codice e compilarlo per ogni possibile sistema operativo su cui andrà ad essere eseguito. Java, al contrario, si basa sull'esecuzione tramite macchina virtuale: praticamente è come se il nostro sistema operativo venisse nascosto al programma che viene quindi eseguito in un ambiente sempre uguale, indipendentemente dal OS di base. Sarà poi compito della JVM tradurre adeguatamente le informazioni in arrivo da e verso il sistema operativo.

Altri punti a favore sono la stretta somiglianza del linguaggio e dei costrutti a quelli del C che ne facilita l'apprendimento e l'uso, oltre alla possibilità di utilizzare ambienti di sviluppo opensource o il semplice Notepad di Windows, riducendo quindi i costi per l'acquisizione di licenze.

1.5 L'interfaccia EIA RS232

Una volta definita la struttura hardware che dovremo esaminare, dobbiamo metterla in grado di comunicare con il sistema di supervisione: i PC mettono a disposizione numerose interfacce, dalla comune USB alla Firewire, ma la scheda S3SKB mette a disposizione solo l'interfaccia RS232 che resta comunque una valida scelta per la semplicità di impiego, a meno di sviluppare tramite i bus di espansione opportuni adattatori per altri standard di comunicazione.

EIA RS-232 [7] *(Electronic Industry Association Recommended Standard 232)* è uno standard EIA equivalente all'europeo CCITT V21/V24 che definisce una interfaccia seriale a bassa velocità per lo scambio di dati tra dispositivi digitali. Nasce negli anni '60 per permettere la comunicazione tra mainframe e terminali classificati come DTE (*Data Terminal Equipment*) attraverso la linea telefonica, impiegando un modem denominato DCE (*Data Communication Equipment*), evolvendosi fino all'ultima revisione del 1997. La versione più diffusa è comunque la RS232c del 1967, che nonostante sia scomparsa dalle applicazioni desktop è ancora largamente utilizzata in campo elettronico e industriale per la comunicazione tra microcontrollori, per la gestione di

dispositivi embedded e in dispositivi relativamente semplici ove non sia richiesta una connessione ad alta velocità.

Il protocollo specifica l'uso di tensioni comprese tra ±5 V e ±15 V: generalmente viene scelto il range ±5 V per le comunicazioni tra circuiti integrati, dato che corrisponde generalmente alla tensione di alimentazione di questi, mentre per la comunicazione verso DTE o DCE si è scelto il range ±12 V che permette una maggiore immunità al rumore. Risulta quindi necessario l'impiego di opportuni dispositivi per interfacciare, ad esempio, un microcontrollore con un PC come il Maxim MAX232 o suoi equivalenti. Da notare che lo standard EIA RS-232 è definito in logica negata: un segnale a tensione positiva corrisponde allo 0 logico (Space) ed uno a tensione negativa all'1 (Mark).

Per incrementare ulteriormente l'immunità ai disturbi elettrici, l'interfaccia elettrica ha una soglia di commutazione di ±3 V ovvero per passare da uno stato ad un altro non è sufficiente arrivare allo zero ma dovrà essere superata la soglia dei 3 V di segno opposto.

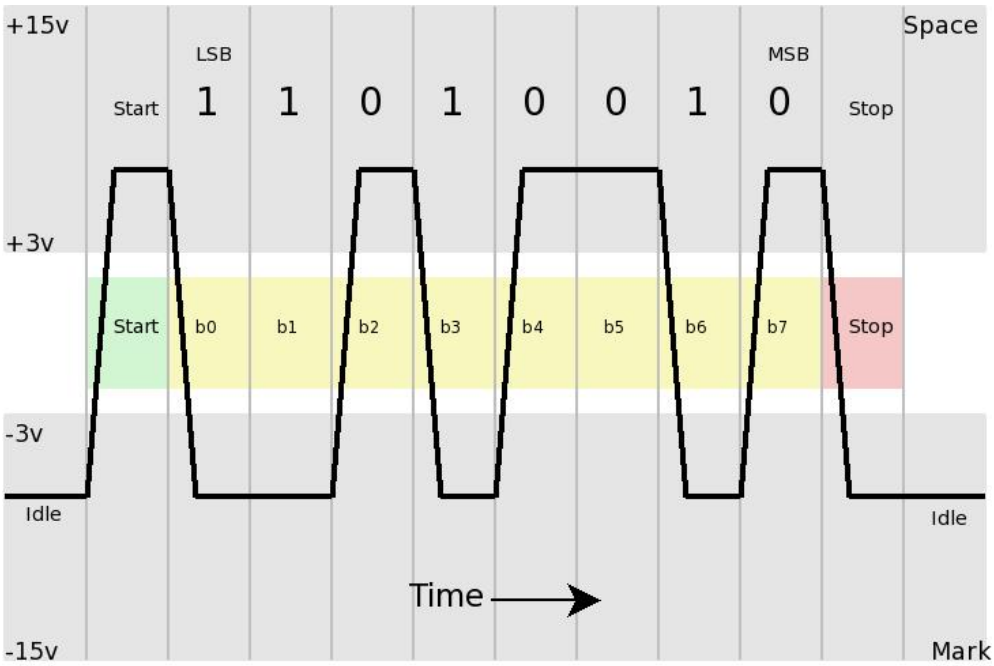

Fig. 1.7 - Segnale RS232

In figura 2.1 è possibile osservare l'invio di dati tramite protocollo RS232 in modalità 8n1: la linea è inizialmente a riposo alla tensione di -12V, quindi viene inviato un bit di start, una sequenza di 8 bit a partire dal meno significativo infine un bit di stop. Da notare l'assenza di bit di parità.

Le possibilità di definizione del pacchetto sono molteplici, scegliendo opportunamente i vari parametri:

- Numero di bit di dati compreso tra 5 e 9 (normalmente 8, 9 non gestibile dai PC);
- Bit di controllo parità di tipo None, Odd (dispari), Even (pari), Mark o Space (normalmente none);
- Numero di bit di stop 1, 1.5 o 2 (normalmente 1).

Da qui le sigle che caratterizzano il tipo di comunicazione: 9600 8n1 indica una velocità di 9600 bps, 8 bit di dati, nessuna parità e 1 bit di stop.

Lo standard definisce anche numerosi segnali di handshake, quindi i bit di start e stop potrebbero essere omessi in un sistema di trasmissione sincrono dove DCE e DTE utilizzano un clock comune e scambiano alcuni flag di stato per segnalare l'intenzione di trasmettere e la possibilità di ricevere. Comunicando però in modalità asincrona, il segnale di clock comune viene a mancare e quindi è necessario poter sincronizzare il sistema ricevente in base all'inizio della ricezione: è qui che entra in gioco il bit di start sul cui fronte di discesa viene inizializzato il clock del ricevente mentre il bit di stop ripristina il livello di idle sulla linea di comunicazione indipendentemente dall'ultimo bit ricevuto.

Vengono inoltre definiti i collegamenti di tipo null-modem impiegati nel caso di comunicazioni tra DTE, anche qui con la possibilità di comunicazione sincrona e asincrona effettuando un handshake completamente o parzialmente locale.

Resta quindi al progettista la scelta di impiegare un tipo di comunicazione sincrona o meno, tenendo presente che nel caso di comunicazione asincrona si richiede solo un clock accurato e due fili di segnale oltre al riferimento di tensione, mentre nel caso sincrono si hanno almeno 9 fili e altrettanti segnali da gestire con notevoli complicazioni circuitali.

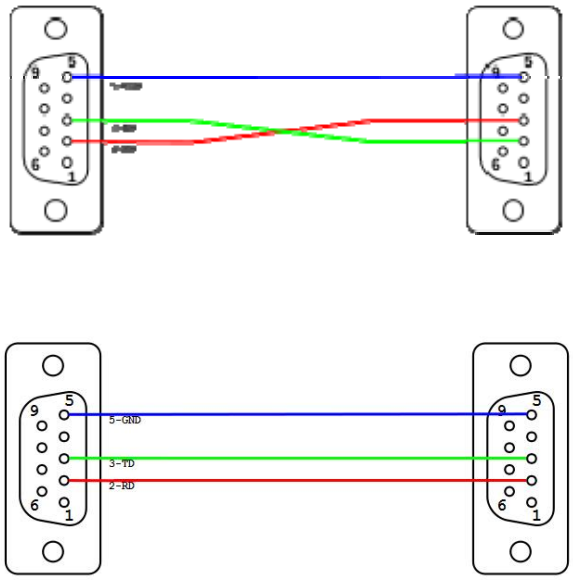

Fig. 1.8 – Collegamento DTE-DTE (null-modem) e DTE-DCE asincrono.

Nel progetto è stato assegnato il ruolo di DTE al PC mentre la scheda S3SKB si comporterà da DCE. In particolare, la comunicazione è asincrona, non avendo a disposizione su scheda tutti i segnali di handshake previsti dallo standard, con velocità di 115200 bps grazie al clock di alta precisione della scheda e con modalità 8n1 per facilitare la gestione dei dati in transito, essendo questi caratteri e quindi byte.

Capitolo 2
Il progetto

2.1 Sezione FPGA

2.1.1 Panoramica

Il circuito implementato su FPGA ha come scopo quello di controllare la comunicazione con il software di supervisione su PC e di fornire a questo la possibilità di controllare il funzionamento del decodificatore.

Si compone essenzialmente di quattro parti (v. fig. 2.1):

- Il picoBlaze che implementa le funzioni di controllo del collaudo e della comunicazione seriale;
- Le memorie FIFO per il salvataggio dei dati per e dal decodificatore;
- Il blocco UARTlite per gestire a livello fisico la comunicazione seriale;
- Alcuni blocchi di interfaccia per adattare i segnali tra picoBlaze e le periferiche.

Al fine di facilitare il debug e di consentire un controllo visivo dello stato del sistema, sono state anche inserite delle unità opzionali che, nel caso si stia collaudando un componente che eccede le risorse disponibili, è possibile escludere per liberare slices.

Tutto il sistema è resettabile tramite la pressione del tasto BTN0, collegato ai segnali di reset dei componenti che lo richiedono.

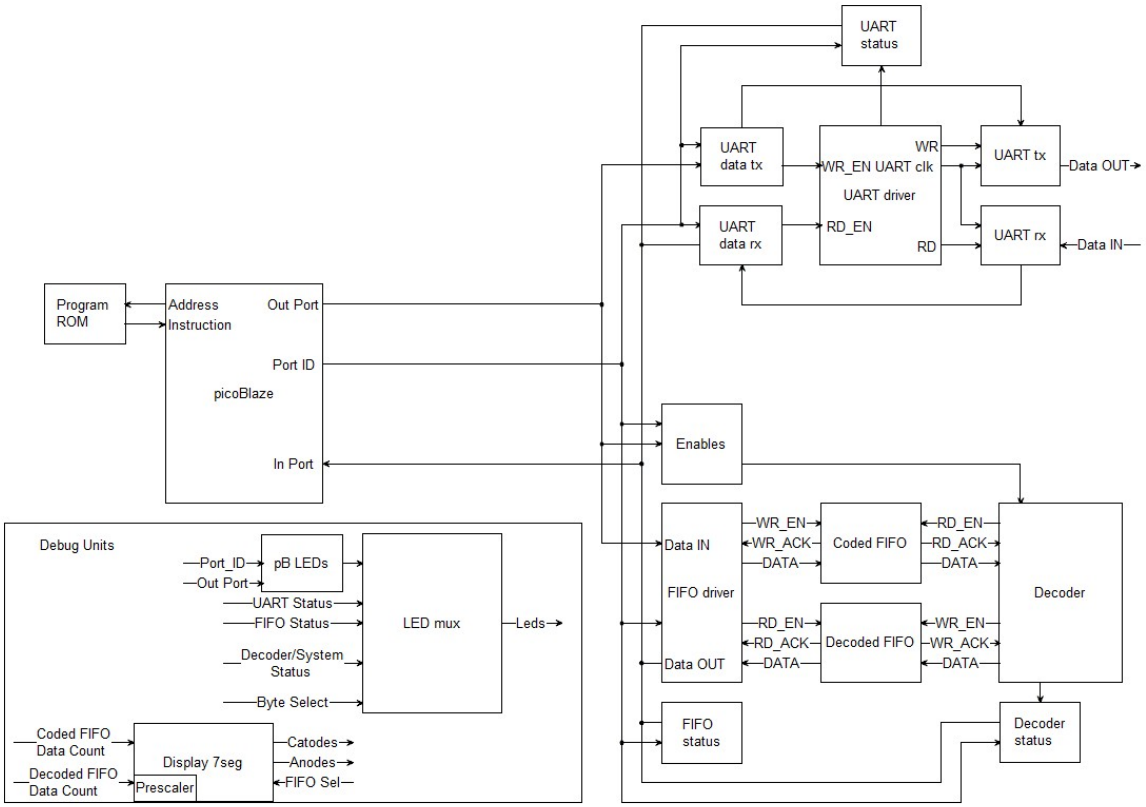

Figura 2.1 – Schema a blocchi del sistema implementato su FPGA (sono stati riportati per semplicità solo i segnali principali).

2.1.2 Il PicoBlaze

Il picoBlaze [8], denominato anche KCPSM3 (“K”onstant Coded Programmable State Machine v3), è un piccolo microprocessore a 8 bit che viene utilizzato principalmente per realizzare macchine a stati finiti “non-time critical” molto complesse utilizzando appena 96 slices, ossia il 5% del’FPGA xc3s200. Viene fornito gratuitamente da Xilinx insieme a esempi di codice, documentazione e agli applicativi necessari all’implementazione del processore in ISE.

Nonostante la sua semplicità mette a disposizione:

- 16 accumulatori a 8 bit;
- 64 byte di memoria dati interna;
- gestione di interrupt;
- 8 linee di uscita;
- 8 linee di ingresso;
- 8 linee per il multiplexing di ingressi e uscite, consentendo di gestire fino a 2048 linee di ingresso e altrettante di uscita;

- segnali di handshake verso le periferiche.

La programmazione del picoBlaze avviene direttamente in assembler, utilizzando quindi il set di istruzioni definito dal suo sviluppatore. Una volta definito il comportamento del processore, il programma viene compilato e convertito in un file VHDL da utilizzarsi in ISE per istanziare la memoria programma che, essendo implementata con l'uso di una block ram, potrà contenere al più 1024 istruzioni.

E' interessante notare come ogni istruzione venga sempre eseguita in due cicli di clock indipendentemente da questa.

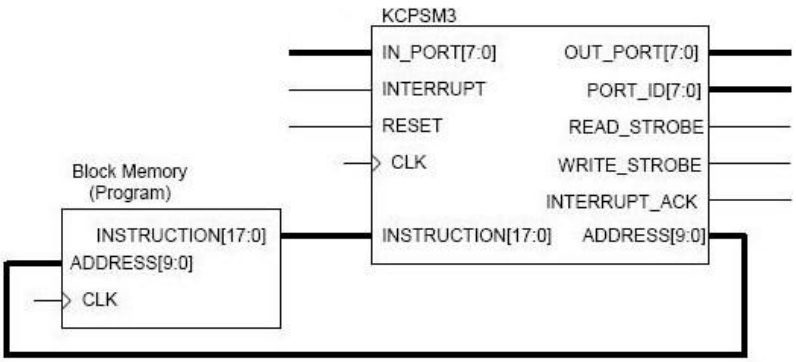

Figura 2.2 – Il PicoBlaze

Nel progetto il picoBlaze è stato inserito principalmente per fornire all'applicazione di supervisione un'interfaccia verso il dispositivo in test, oltre che per gestire un handshake software durante la comunicazione seriale rispondendo in base allo stato del sistema ai comandi ricevuti dal PC. Tale blocco sarebbe stato possibile implementarlo anche con logica combinatoria e FSM, ma avrebbe avuto un'occupazione di risorse tale da impedire l'implementazione di decodificatori sull'FPGA.

2.1.3 Sezione I/O PicoBlaze

Nel paragrafo precedente si è visto come il picoBlaze abbia 8 linee di ingresso e 8 linee di uscita: dovendo gestire vari segnali nel progetto, si è reso necessario realizzare degli opportuni blocchi di multiplexing e memorizzazione per gli ingressi e le uscite.

Relativamente agli ingressi, si tratta di un multiplexer a 8 bit di indirizzo e 8 bit di uscita: il segnale di *port_id* seleziona l'ingresso che si desidera leggere che viene riportato e tenuto stabile verso il picoBlaze al primo fronte di clock. Anche se il bus di indirizzo viene condiviso con le linee di uscita, non si verificano problemi come la lettura di dati errati in quanto ad ogni istruzione di lettura viene aggiornato l'indirizzo del dato desiderato. Essendo un multiplexer senza memorizzazione non è stato necessario inserire un reset asincrono del blocco. Inoltre, i bus di input e output sono separati, non richiedendo quindi l'assegnazione esclusiva degli indirizzi a un porto di ingresso o di uscita e la necessità di porre in alta impedenza le linee dati verso il picoBlaze dopo ogni operazione.

Non è stato necessario controllare anche il segnale *read_strobe* in quanto il bus di indirizzo resta stabile per tutta la durata dell'operazione di lettura.

Per ridurne l'occupazione di risorse, solo gli indirizzi effettivamente corrispondenti a un porto di ingresso sono stati implementati: in tutti gli altri casi, il bus assumerà il valore 0 su ogni linea. Nel caso si debba poter leggere ulteriori ingressi, sarà necessario modificare il componente inserendo un ulteriore bus di ingresso con il relativo indirizzo all'interno del multiplexer.

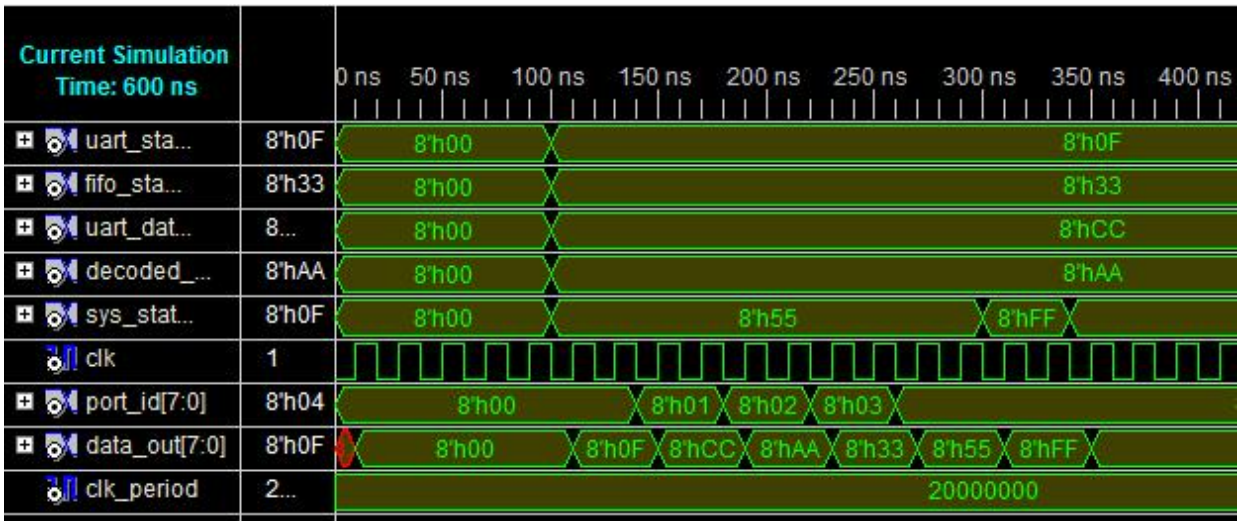

Figura 2.3 – Simulazione del blocco di ingresso

La sezione di uscita è composta da registri a 8 bit nei quali la scrittura viene abilitata solo al verificarsi contemporaneo di:

- segnale di *port_id* corrispondente al parametro *generic* specificato in fase di dichiarazione del componente;

- impulso di *write_strobe* dal picoBlaze.

In particolare, la seconda condizione impedisce una scrittura accidentale sul registro nel caso in cui si vada a leggere un ingresso con lo stesso indirizzo.

Il blocco è dotato di reset asincrono, in modo tale che sia possibile riportarlo allo stato iniziale senza dover eseguire nel programma del controllore una routine di azzeramento delle uscite.

In caso sia necessario realizzare ulteriori uscite, basterà istanziare un ulteriore componente assegnando al campo generic un indirizzo disponibile.

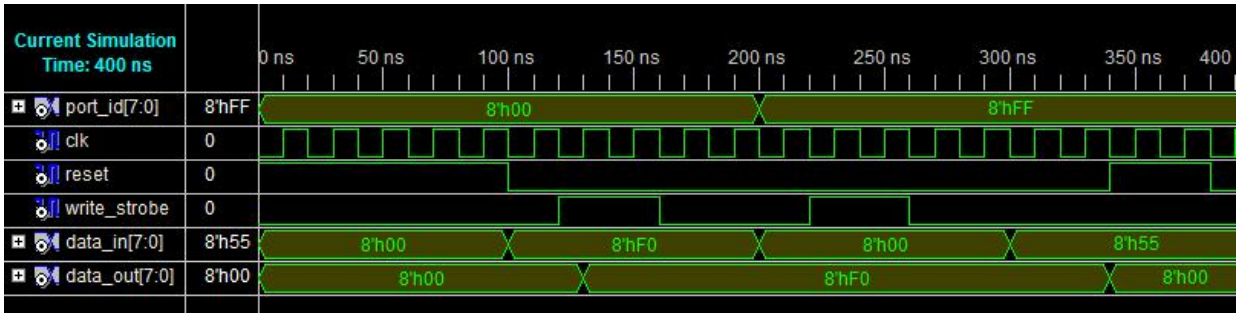

Figura 2.4 – Simulazione di un blocco di uscita con indirizzo 0X00.

Nelle tabelle seguenti vengono riportati i porti di ingresso e di uscita utilizzati nel progetto con il relativo indirizzo espresso in formato esadecimale e la funzione dei singoli bit dei porti.

USCITE		INGRESSI	
Indirizzo	Descrizione	Indirizzo	Descrizione
00	UART DATA TX	00	UART STATUS
01	pB LED	01	UART DATA RX
02	CODED FIFO LOW BYTE	02	DECODED FIFO DATA
03	CODED FIFO HIGH BYTE	03	FIFO STATUS
04	ENABLE BYTE	04	SYS STATUS (DECODER)

COMPOSIZIONE BYTES DI STATO E CONTROLLO				
BIT	UART STATUS	FIFO STATUS	SYS STATUS	ENABLE BYTE
0	RX BUFFER DATA PRESENT	DECODED FIFO WR ACK	DECODER BUSY	DECODER ENABLE
1	RX BUFFER HALF FULL	DECODED FIFO VALID	X	DECODED FIFO RD EN
2	RX BUFFER FULL	DECODED FIFO FULL	X	CODED FIFO WR EN
3	TX BUFFER HALF FULL	DECODED FIFO EMPTY	X	X
4	TX BUFFER FULL	CODED FIFO WR ACK	X	X
5	X	CODED FIFO VALID	X	X
6	X	CODED FIFO FULL	X	X
7	X	CODED FIFO EMPTY	X	X

2.1.4 Sezione UART

Lo UART [9] (Universal Asyncronous Receiver Transmitter) è un dispositivo generico che converte un flusso di dati dal formato parallelo a seriale asincrono e viceversa. Insieme con il picoBlaze ne viene fornita una versione denominata “UART lite” che implementa l’interfaccia di comunicazione tra il picoBlaze e il canale RS-232c in modalità 8n1.

Le caratteristiche principali sono:

- buffer FIFO verso il canale da 16 byte;
- buffer FIFO verso il pB da 16 byte;
- velocità massima fino a 10 Mb/s tra FPGA Xilinx;
- occupazione di 40 slices;

La comunicazione avviene in modalità asincrona e richiede pertanto la generazione di un segnale di clock locale in funzione della velocità di trasmissione che si sincronizzi su un evento certo della comunicazione, in modo tale da poter inviare e ricevere correttamente i dati. La sincronizzazione tra ricevitore e trasmettitore avviene sul fronte del bit di start e il campionamento del bit si ha a metà del tempo di simbolo, ossia dove il segnale è più stabile. Ad ogni impulso del clock, il ricevitore legge lo stato della linea, memorizzando i bit a partire dal LSB quindi al termine del dato verifica il bit di stop per avere conferma della corretta trasmissione. Sincronizzandosi sul primo bit, al termine

della trasmissione si dovrà avere una tolleranza massima sulla temporizzazione pari a mezzo tempo di simbolo: utilizzando una comunicazione 8n1 che impiega 10 bit, si ha che la tolleranza massima è del 5%, facilmente ottenibile nei sistemi digitali odierni.

Il componente UART lite richiede che venga applicato un segnale di clock impulsivo con frequenza pari a 16 volte il bit rate desiderato: è facile quindi determinare il numero di cicli di clock necessari tramite la relazione $count = \frac{f_{osc}}{16 \times bps}$ arrotondando all'intero più vicino.

Viene messa a disposizione una ricca interfaccia che comprende:

- segnali di stato del buffer di ricezione;
- segnali di stato del buffer di trasmissione;
- strobes per la lettura e l'invio di dati;
- reset per l'azzeramento dei buffer.

Per generare i segnali di lettura e scrittura, oltre al segnale di sincronia, è stato realizzato un opportuno driver.

Il segnale di sincronia è un contatore che raggiunto il valore specificato nei parametri generic porta a livello alto l'uscita, per poi resettarsi al successivo ciclo di clock.

I comandi di lettura e invio vengono realizzati utilizzando come base il registro di uscita per il picoBlaze: relativamente al segnale di lettura è stato eliminato il registro in questo contenuto avendo già il buffer del componente UARTlite che si occupa di tenere stabile il dato, mentre è stato inserito un controllo sul *read_strobe* al fine di evitare letture indesiderate dal buffer nel caso si stia scrivendo su un registro di uscita con lo stesso indirizzo.

Allo stesso modo viene realizzato il segnale di scrittura impiegando ovviamente il segnale *write_strobe* ed eliminando anche qui il registro di uscita, dato che il buffer richiede che il dato sia stabile per un solo ciclo di clock mentre il pB garantisce due cicli di clock.

Come per il registro di uscita, se non si ha corrispondenza tra gli indirizzi (specificati come parametri generic in ISE) e il *port_id* non si ha generazione degli impulsi di comando.

Dato che il driver è di natura assolutamente combinatoria e non vi sono registri con dati, non è stato ritenuto necessario implementare un reset asincrono.

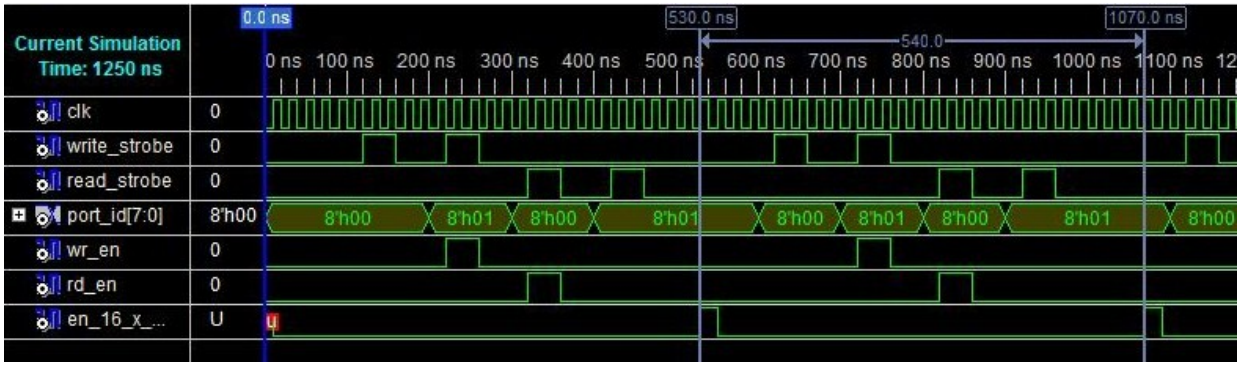

Figura 2.5 – Simulazione del driver UART per comunicazione a 9600 bps

2.1.5 Le FIFO e l'interfaccia con il pB

I dati generati dall'applicazione di supervisione e i relativi risultati della decodifica, vengono salvati in memorie di tipo FIFO in quanto non è possibile effettuare una decodifica continua con l'invio e ricezione di dati tramite l'interfaccia seriale a causa del limitato bit rate di quest'ultima: infatti, anche se non si implementasse alcun controllo di flusso, si avrebbe un bit rate massimo pari a 115200 bit/s che corrisponde a 14.4 kB/s mentre il decodificatore può raggiungere velocità nell'ordine di 1 MB/s e come conseguenza non sarebbe possibile effettuare una stima delle prestazioni del decoder.

Impiegando le memorie FIFO, invece, è possibile bufferizzare il transito dei dati mettendone una quantità consistente e nota a disposizione del dispositivo in esame, avviare l'elaborazione e scaricare quindi i risultati senza il rischio di saturare il canale di comunicazione. Avendo a disposizione sull'FPGA XC3S200 11 block ram libere e volendo sfruttarle al massimo per ottenere la maggior quantità di dati memorizzabili possibile, è stato scelto di impiegare 4 block ram per ciascuna memoria. Tenendo presente che queste sono memorie in grado di contenere 1024 parole da 16 bit e che il rapporto di codifica è 1:2, è possibile quindi memorizzare un totale di 4096 byte sia da che verso il decoder.

L'impiego delle block ram avviene tramite opportuni componenti generati dal CoreGenerator contenuto nel pacchetto di sviluppo ISE [10]: questo permette di impiegare delle interfacce ottimizzate per il tipo di FPGA a disposizione e di ridurre considerevolmente l'occupazione di risorse. Tra le varie opzioni, è possibile specificare:

- Tipo di implementazione (block ram, ram distribuita, shift register);

- Dimensione delle parole da memorizzare;
- Profondità della memoria;
- Segnali di handshake opzionali quali *write ack, data valid, FIFO half full,* etc. ;
- Conteggio delle parole contenute nella memoria;
- Tipo di reset.

Volendo assicurarsi l'integrità dei dati, è stato scelto di utilizzare i segnali opzionali di *write acknowledge* e di *data valid*: il primo assicura che la scrittura di un dato è andata a buon fine e quindi è possibile eseguire un'altra operazione di scrittura, mentre il secondo garantisce che il dato presente nel registro di uscita è valido.

A fini di debug, è stato anche specificato il conteggio delle parole contenute nelle FIFO: tali valori sono rappresentati all'utente in formato esadecimale sui display a sette segmenti della S3SKB. Come per le unità opzionali, anche questi contatori possono essere eliminati nel caso servano maggiori risorse a disposizione del componente in esame.

Anche per queste periferiche è stato necessario realizzare un driver ad hoc che permettesse di adattare i segnali di handshake al funzionamento del picoBlaze.

I segnali verso le memorie devono essere delle durata di un impulso di clock onde evitare letture o scritture multiple: vengono quindi generati tramite una FSM che porta a livello logico alto l'uscita per un ciclo di clock e quindi resta in attesa fino a quando il segnale dal picoBlaze torna a zero, dove si resetta e torna allo stato iniziale.

I segnali verso il picoBlaze risultano impulsivi e quindi non campionabili dal controllore: l'adattamento avviene tramite un flip-flop al cui set è collegato il segnale di conferma dalla memoria, mentre al reset si ha il segnale di comando proveniente dal pB. In questo modo, il segnale di *valid* o di *write_ack* verso il pB viene mantenuto stabile per almeno due cicli di clock, permettendo l'operazione di lettura degli ingressi, mentre viene resettato all'avvento di una richiesta di lettura o scrittura impedendo una falsa interpretazione dello stato delle FIFO.

E' stato anche implementato un reset asincrono in modo da riportare il componente alle condizioni iniziali.

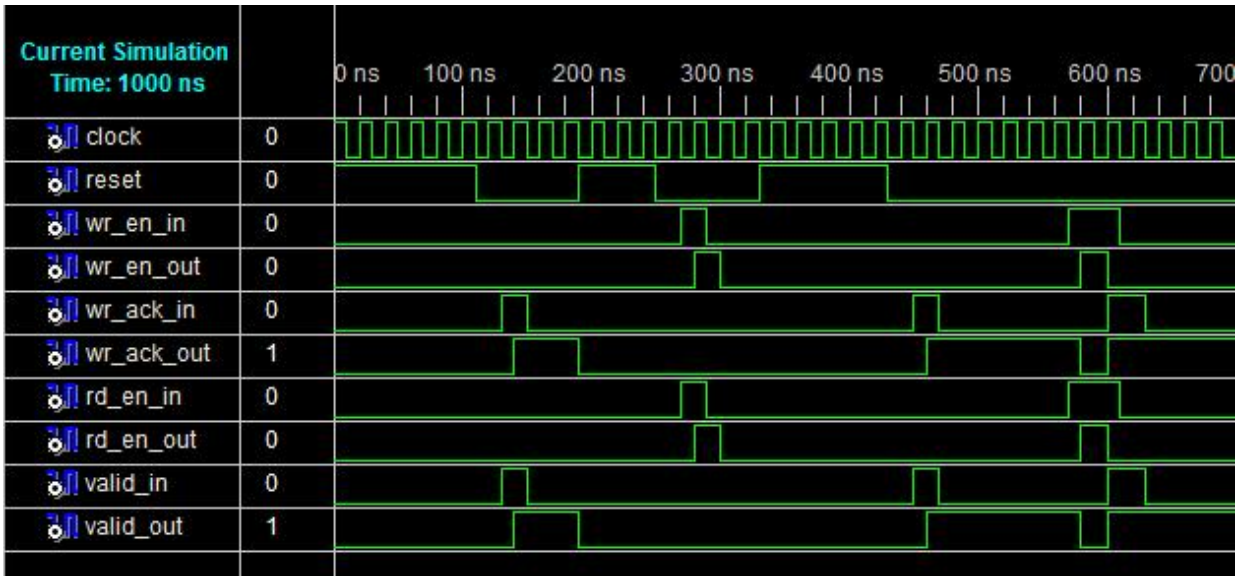

Figura 2.6 – Simulazione del driver FIFO

2.1.6 Unità di debug opzionali

Il debug di un sistema integrato risulta difficile da eseguire principalmente per l'impossibilità di osservare ciò che accade durante il funzionamento. Al fine di permettere l'osservazione dello stato del sistema, sono state realizzate delle unità che si occupano di visualizzare alcuni parametri sui display e sui LED a disposizione sulla S3SKB.

In particolare, i quattro display a 7 segmenti sono impiegati per la visualizzazione del numero di parole contenute nelle FIFO in formato esadecimale, mentre gli 8 led mostrano la condizione dei byte di stato del sistema.

Nella tabella seguente vengono mostrate le impostazioni degli switches e i parametri corrispondenti.

SW7	SW1	SW0	Display 7seg	LED
0	X	X	Decoded FIFO	X
1	X	X	Coded FIFO	X
X	0	0	X	pB LED
X	0	1	X	UART Status
X	1	0	X	FIFO Status
X	1	1	X	Enables *
* Bit 7 corrispondente allo stato del decoder.				

2.1.7 Il Programma caricato nel PicoBlaze

Il cuore di tutta la sezione FPGA è il firmware di gestione caricato all'interno del picoBlaze. Verranno di seguito illustrate le funzioni implementate.

MAIN

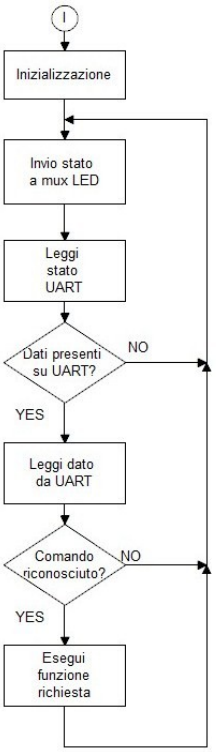

Figura 2.7 – Diagramma di flusso della procedura "main"

Inizializza il processore e manda in esecuzione le funzioni richieste.

In Fig. 2.7 è possibile vederne il funzionamento schematico: dopo una fase di inizializzazione dove vengono disabilitati gli interrupt, si entra nel ciclo di interrogazione della seriale dove si verifica la presenza di dati nel buffer di ricezione leggendo il byte di stato della UART. Nel caso ve ne siano, viene letto il primo byte e tramite una struttura if-then si richiama la funzione richiesta; se il comando non è implementato, il byte viene ignorato e si procede nuovamente con il ciclo di interrogazione.

STATUS

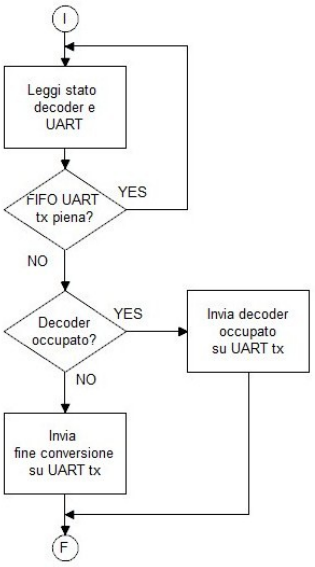

Figura 2.8 – Diagramma di flusso della procedura "Status"

Comunica all'applicazione di supervisione lo stato del decoder. In fig. 2.8 lo schema di funzionamento.

Dopo aver letto il byte di stato del decoder e aver determinato la possibilità di inviare un byte al PC, viene testato il bit di "decoder busy": in base al risultato, viene inviato un byte che indica la condizione di lavoro del decoder. In questo modo, l'applicazione di controllo può determinare se il decodificatore ha terminato l'elaborazione dei dati e quindi stimare il tempo di elaborazione e richiedere i risultati della decodifica.

READY TO SEND

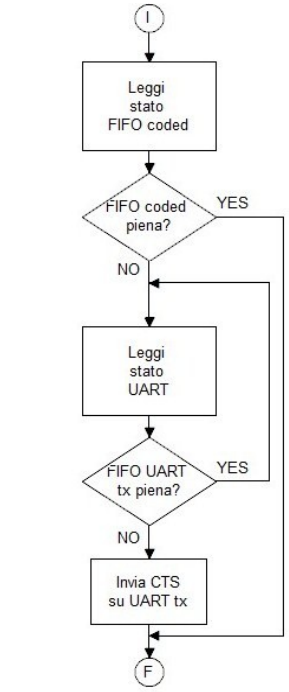

Figura 2.9 – Diagramma di flusso della procedura "RTS"

Utilizzata per determinare la possibilità di invio di dati dal PC verso il decoder. In fig. 2.9 lo schema di funzionamento.

Viene letto il byte di stato delle FIFO e controllato il bit "Coded FIFO full": se risulta possibile il salvataggio di dati, viene inviato un byte denominato CTS (Clear to Send) all'applicazione di gestione che può quindi procedere con l'invio di dati. Nel caso in cui la memoria sia piena, il comando viene ignorato e toccherà all'applicazione di controllo determinare il timeout della richiesta.

L'invio del comando avviene dopo aver verificato la possibilità di inviare un byte perché se si avesse il buffer pieno e si tentasse di scrivervi un ulteriore byte, questo verrebbe ignorato generando una condizione di timeout nell'applicazione di supervisione.

ECHO

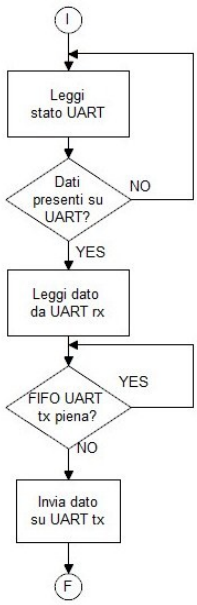

Figura 2.10 – Diagramma di Flusso della procedura "Echo"

Procedura per verificare il collegamento e la corretta impostazione della connessione seriale tra FPGA e PC.

Richiede come parametro un byte che dovrà essere presente nel buffer di ricezione della porta seriale.

In fig. 2.10 il diagramma di flusso.

Viene interrogato lo stato della UART fino a che non si ha almeno un byte presente in ricezione. Una volta letto, questo viene reinviato all'applicazione di controllo effettuando il consueto controllo sul buffer di invio.

Se il carattere ricevuto dal PC corrisponde a quello inviato, allora la comunicazione è correttamente stabilita. Nel caso di impostazioni errate, si possono ricevere caratteri non corrispondenti o non riceverne affatto: sarà compito dell'applicazione di supervisione determinare tramite timeout della connessione l'errata impostazione.

ENABLE/DISABLE DECODER

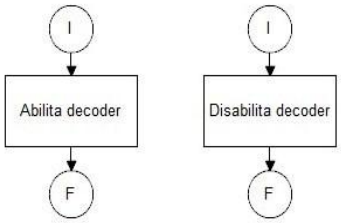

Figura 2.11 – Diagrammi di flusso procedure "Enable" e "Disable"

Procedure per abilitare e disabilitare il funzionamento del decodificatore.

Impostano rispettivamente a 1 e 0 il bit di "decoder enable" nel byte "enables".

DOWNLOAD

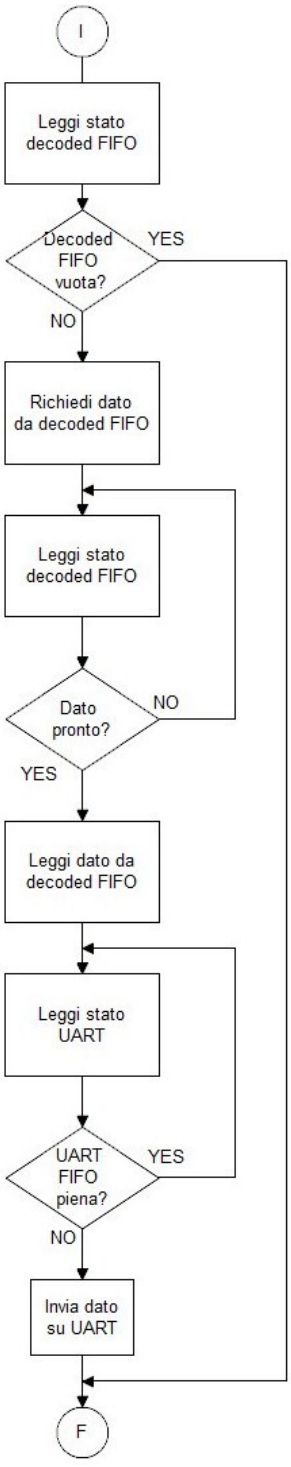

Figura 2.12 – Diagramma di flusso della procedura "Download"

Procedura per la lettura di un byte dalla FIFO decoded e l'invio al PC. In fig. 2.12 il funzionamento schematizzato.

Si interroga lo stato delle FIFO per determinare se vi sono dati da leggere: con l'esito negativo si termina l'esecuzione.

Viene richiesto un dato alla FIFO e viene interrogato lo stato fino ad avere conferma tramite il bit valid che il dato è corretto, che viene quindi acquisito dal processore.

Il byte così ottenuto viene quindi inviato all'applicazione di controllo effettuando preventivamente un controllo sullo stato del buffer di invio dell'interfaccia UART.

DATA

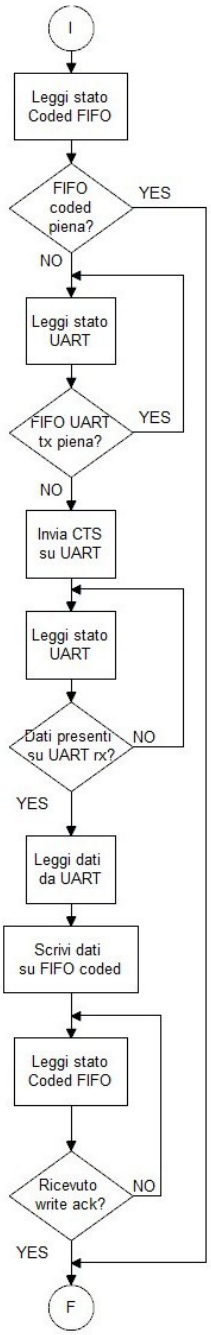

Figura 2.13 - Diagramma di flusso della procedura “Data”

Procedura per la ricezione di dati da PC, richiede come parametri due byte. In fig. 2.13 ne viene rappresentato il funzionamento.

Viene effettuato un test sullo stato della FIFO coded per verificare la possibilità di salvataggio dei dati: in caso l’esito sia negativo, vengono ignorati i dati e terminata l’esecuzione; in caso contrario, viene confermata la richiesta con l’invio del codice CTS.

Si esegue quindi un polling sullo stato della UART per ricevere il primo byte, che viene salvato nel registro di ingresso della FIFO, come da specifica RS232, come meno significativo. Tali operazioni vengono ripetute anche per il secondo byte, che viene quindi salvato come byte più significativo.

Viene avviata la scrittura nella FIFO e si attende la conferma leggendo il bit “Write Ack” nello stato delle memorie. Ottenuta questa, viene terminata l’esecuzione.

2.2 Sezione PC

2.2.1 Panoramica

L'applicazione di supervisione svolge il compito di generare i dati in maniera casuale, codificarli e inviarli tramite porta seriale al picoBlaze per poi attivare il decodificatore. Una volta terminata l'elaborazione, scarica i risultati e li confronta con i dati attesi per determinare l'esito della decodifica e generare i file di report.

Il programma si compone di un metodo "main" che richiama al suo interno i metodi creati per svolgere le singole funzioni necessarie: tale approccio è stato scelto per rendere modulare l'applicazione e permetterne un agevole sviluppo. Inoltre è possibile adattare il funzionamento a un qualsiasi decodificatore modificando solo alcuni metodi, come quelli relativi alla codifica e generazione dei dati, mantenendo inalterati i restanti.

Si utilizza una opportuna libreria per la comunicazione seriale e viene impiegato un controllo di flusso software per gestirla.

Nella fase di debug, è stato utile simulare la sezione FPGA tramite il programma Hyperterminal fornito insieme con il sistema operativo Windows XP. Per collegare l'applicazione a Hyperterminal è stato necessario impiegare un ulteriore applicativo freeware denominato "com0com" che emula due porte seriali collegate tra loro.

2.2.2 Libreria RXTX

La sezione FPGA ha necessità di comunicare con il PC tramite porta seriale ma Sun Microsystems, la sviluppatrice di Java, ha deciso di discontinuare le librerie di supporto alla comunicazione seriale e parallela [11]. Risultava quindi necessario migrare verso un altro linguaggio di programmazione come C o MatLab. Fortunatamente, nella ricerca di una soluzione al problema, è stato individuato un gruppo di sviluppatori che ha ridefinito le librerie di comunicazione rendendo quindi possibile l'utilizzo di Java.

La libreria RxTx [12] viene fornita gratuitamente e con licenza opensource tramite il sito degli sviluppatori. Implementa la gestione della comunicazione seriale mettendo a disposizione un vasto gruppo di metodi per il controllo della comunicazione e per ottenere degli oggetti di tipo

InputStream e OutputStream, rispettivamente per inviare e ricevere dati, che puntano la porta seriale specificata. Viene supportata anche la comunicazione full duplex tramite l'impiego di threads.

La libreria è inoltre fornita dei file necessari per interfacciarsi verso il sistema operativo, qualsiasi esso sia, permettendo l'uso dell'applicazione di controllo indipendentemente dal sistema di base.

2.2.3 Struttura dell'applicazione di supervisione

L'applicazione si compone di un metodo "main" che richiama al suo interno i vari metodi necessari alle operazioni. Il funzionamento è procedurale (v. fig. 2.14) e quindi si poteva omettere la realizzazione di metodi da richiamare, ma è stato preferito un programma a funzioni per facilitare lo sviluppo e il debug di ogni singola sezione dell'applicazione oltre a rendere più leggibile il codice e a permettere modifiche più agevoli.

Verranno di seguito illustrati i metodi realizzati, indicandoli con la propria firma.

```
public static void main ()
```

Esegue la funzione di interfaccia verso l'utente, inizializza la porta seriale, utilizza i metodi della classe per eseguire il controllo del collaudo e redige i file di report del test contenenti le statistiche sintetiche del test e i dati che sono stati inviati, ricevuti e la relativa maschera di errore.

Non richiede parametri.

```
private static void send (InputStream datain, OutputStream dataout, byte[] d, int num_byte)
```

Metodo per l'invio dei dati verso il picoBlaze. Implementa un controllo di flusso software per la comunicazione avvalendosi dei metodi della classe.

Impiega i seguenti parametri:

- datain: InputStream corrispondente alla porta seriale dal quale si ottengono i dati dall'FPGA;
- dataout: OutputStream corrispondente alla porta seriale sul quale si scrivono i dati da inviare all'FPGA;
- d: array dei byte codificati da inviare per l'elaborazione;
- num_byte: lunghezza in byte della parola in chiaro.

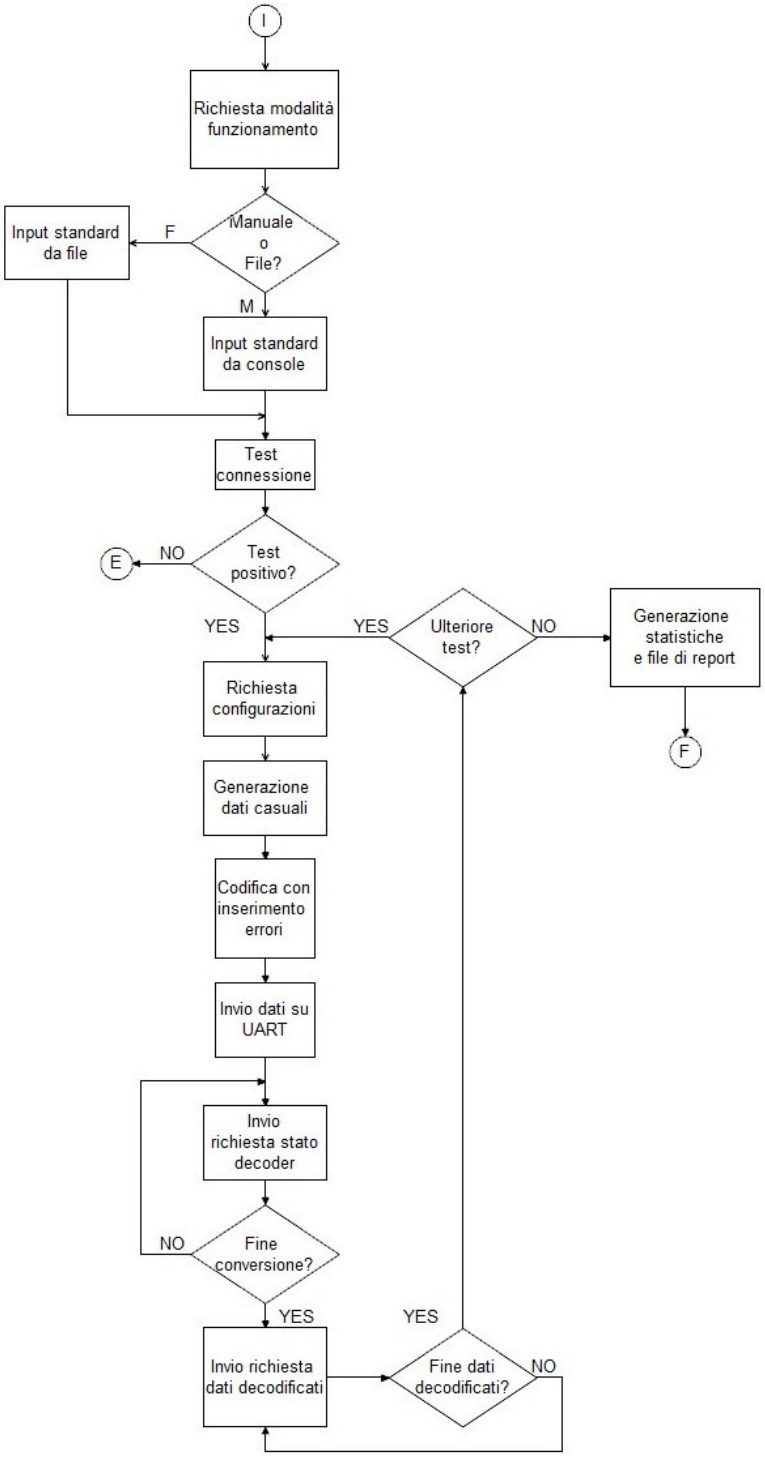

Figura 2.14 – Diagramma di flusso dell'applicazione di supervisione

`private static byte[] waitForSignal(InputStream datain, OutputStream dataout, byte signal)`

Metodo per la ricezione dall'FPGA dei byte di controllo di flusso. Termina l'esecuzione alla ricezione del byte specificato o lancia un'eccezione in caso di timeout. Restituisce un array di byte in cui si ha il buffer della porta seriale per permettere ulteriori controlli.

Impiega i seguenti parametri:

- datain: InputStream corrispondente alla porta seriale dal quale si ottengono i dati dall'FPGA;
- dataout: OutputStream corrispondente alla porta seriale sul quale si scrivono i dati da inviare all'FPGA;
- signal: byte di controllo di flusso da attendere.

`private static byte[] receive (InputStream datain, OutputStream dataout, int n, int num_byte)`

Metodo per la ricezione dei dati elaborati dall'FPGA. Implementa un controllo di flusso software per la comunicazione avvalendosi dei metodi della classe. Restituisce al chiamante un array di byte contenente i dati ricevuti.

Impiega i seguenti parametri:

- datain: InputStream corrispondente alla porta seriale dal quale si ottengono i dati dall'FPGA;
- dataout: OutputStream corrispondente alla porta seriale sul quale si scrivono i dati da inviare all'FPGA;
- n: numero di parole da ottenere;
- num_byte: lunghezza in byte della parola in chiaro.

`private static void end_test(InputStream datain, OutputStream dataout)`

Metodo per l'interrogazione dello stato del decoder. Effettua un polling tramite l'invio del comando di interrogazione e ottiene il codice di stato: se risulta che il decoder è occupato ripete l'operazione. Termina l'esecuzione quando il picoBlaze comunica il termine dell'elaborazione dei dati.

Impiega i seguenti parametri:

- datain: InputStream corrispondente alla porta seriale dal quale si ottengono i dati dall'FPGA;
- dataout: OutputStream corrispondente alla porta seriale sul quale si scrivono i dati da inviare all'FPGA.

`private static String formatString(String s, int dim, char align, char filler)`

Metodo per la formattazione delle stringhe di testo impiegate nella realizzazione del report a video. Restituisce una stringa formattata opportunamente.

Impiega i seguenti parametri:

- s: stringa di testo da formattare;
- dim: numero di caratteri disponibili su cui formattare la stringa s;
- align: allineamento della stringa (R=destra, L=sinistra, C=centrato, default R);
- filler: carattere da utilizzare come riempitivo per l'allineamento.

`private static double[] generate (int n)`

Metodo per la generazione dei dati casuali. Restituisce un array di double contenente il valore numerico dei byte da codificare. E' possibile specificare la generazione di soli dati alfanumerici rimuovendo il commento all'interno del codice.

Il parametro n specifica la lunghezza dell'array.

`private static byte[] encode(double[] d, boolean[][] noise, int num_byte)`

Metodo per la codifica dei dati da inviare all'FPGA. Restituisce un array di byte contenente le parole codificate su cui è stata applicata la maschera di errore che simula il rumore nella comunicazione.

Impiega i seguenti parametri:

- d: array di double contenente le parole in chiaro;
- noise: matrice di boolean per l'applicazione del rumore alle parole codificate;
- num_byte: lunghezza in byte della parola in chiaro.

`private static byte[] doubleToByte (double[] d)`

Metodo di conversione da array di double ad array di byte. Restituisce l'array di byte con la conversione.

Il parametro d contiene di dati da convertire.

`private static boolean[] doubleToBit(double d)`

Metodo di conversione da double a array di boolean. Restituisce la rappresentazione binaria di un double sotto forma di array di boolean.

Il parametro d è il valore da convertire.

`private static boolean[] hexToBit(char c)`

Metodo di conversione da esadecimale a bit. Restituisce la rappresentazione binaria di una cifra esadecimale sotto forma di array di boolean.

Il parametro c è la cifra da convertire.

`private static byte[] bitToByte(boolean[] bit)`

Metodo di conversione da stream binario a array di byte. Restituisce un array di byte contenente la conversione dei sottoarray di boolean.

Impiega il parametro bit che contiene lo stream binario sotto forma di array di boolean. Si assume che ogni byte occupi 8 bit.

2.2.4 Uso dell'applicazione

La procedura per eseguire il collaudo di un decodificatore richiede pochi semplici passaggi:

1. Programmare la S3SKB con il bitstream contenente il decoder e l'interfaccia di collaudo;
2. Collegare la S3SKB alla porta seriale del PC su cui si eseguirà il test;
3. Alimentare la S3SKB;
4. Eseguire il programma di supervisione seguendo le istruzioni a video.

Verranno richieste alcune configurazioni, dopo di che il test sarà effettuato in maniera totalmente automatica. In particolare sarà richiesto all'utente di inserire nell'ordine:

- La modalità di funzionamento, se manuale o tramite file di configurazione generato dal programma "ViterbiConfig";
- Il numero di byte che compone le singole parole in chiaro;
- Il numero della porta COM da utilizzare;
- La velocità di comunicazione;

- Il numero di parole da generare per il test (in caso si ecceda viene rettificato automaticamente);
- La modalità di applicazione degli errori (“manuale” prevede l’inserimento di una unica maschera uguale per tutti gli invii, “file” utilizza una maschera unica per tutti gli invii specificata in un opportuno file, “random” genera un array di maschere casuali con un numero fisso di bit errati per invio).

Vengono quindi effettuate tutte le operazioni necessarie al collaudo, notificando all’utente l’avanzamento del test.

Una volta terminata l’esecuzione, è possibile effettuare ulteriori test oppure terminare l’applicazione. In quest’ultimo caso viene stampato a video un report sintetico del collaudo e sono messi a disposizione due file, uno con il report sintetico e uno contenente i dati inviati, ricevuti e la maschera di errore applicata, per eventuali ulteriori elaborazioni dei risultati.

Viene di seguito riportato un esempio dei dati contenuti nel file di report per un invio di parole da 2 byte: i primi due dati sono la stringa inviata e ricevuta in formato Unicode, mentre il terzo è la maschera di errore applicata al dato codificato. I dati sono separati da punti e virgola, mentre i singoli caratteri sono divisi con virgole. Nella maschera di errore, un “1” rappresenta il bit errato mentre uno “0” un bit non modificato. La scelta di rappresentare i caratteri in formato Unicode nasce per poter permettere il trattamento di testi tramite il decodificatore, identificando in maniera univoca un qualsiasi carattere prescindendo dalla tabella codici in uso nel sistema operativo.

90,236;90,236;00000001000100000001000000001000;

2.2.5 Il programma ViterbiConfig

La procedura di collaudo normale richiede la presenza di un operatore al PC che fornisca all’applicazione le specifiche del collaudo da effettuare. Per svolgere in maniera completamente automatica test gravosi, come ripetere l’invio di dati variando il BER (Bit Error Rate) ad ogni ciclo, è stata scritta una opportuna applicazione per la definizione di un file di configurazione.

Viene richiesta la configurazione dell’interfaccia seriale, il numero di byte in chiaro per parola, il numero di parole per singolo test, il numero di ripetizioni del singolo test, il BER minimo e massimo. La generazione degli errori è impostata di default in modalità random. Anche qui, con semplici variazioni al codice, è possibile aumentare le opzioni per il collaudo.

Terminato l'inserimento dei parametri, sarà sufficiente eseguire il programma di collaudo specificando l'esecuzione in maniera automatica.

E' possibile vedere un esempio della procedura da seguire in figg. 2.15 e 2.16.

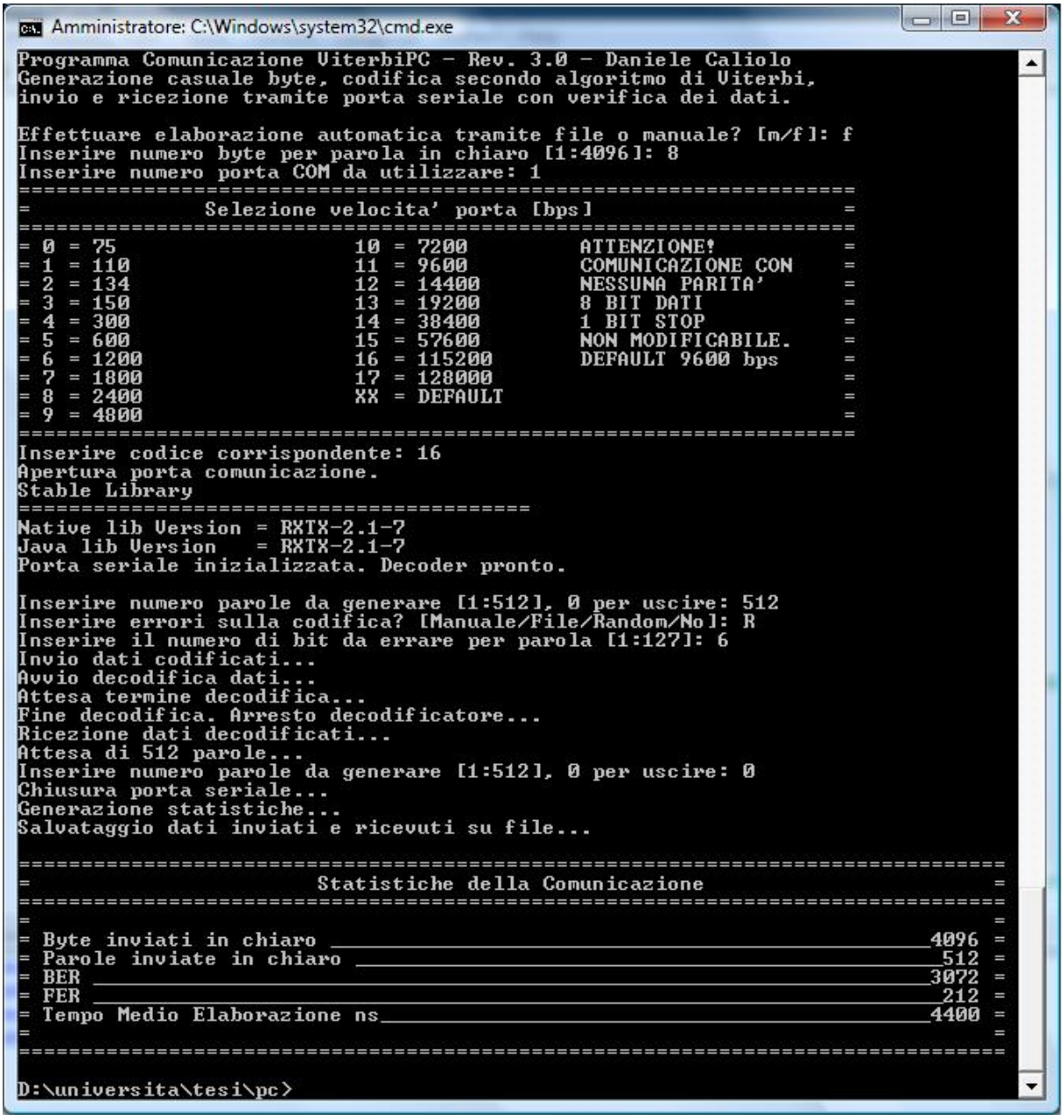

Figura 2.15 – Esempio di esecuzione della procedura di collaudo. Da notarsi l'esecuzione automatica tramite file di configurazione

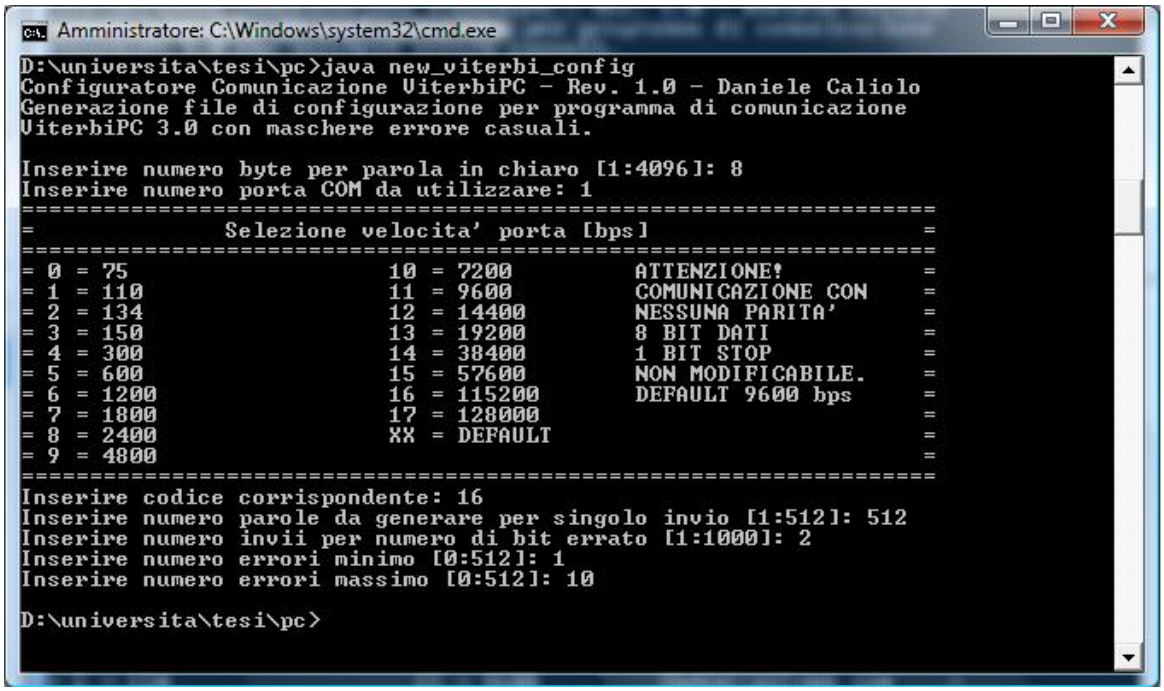

Figura 2.16 – Esempio di esecuzione del programma ViterbiConfig

2.3 Collaudo del decoder "Viterbi"

Il dispositivo in esame implementa un decodificatore secondo l'algoritmo di Viterbi [13]. Contiene al suo interno l'interfaccia per la gestione delle memorie FIFO e mette a disposizione un bit di stato e un segnale di chip enable.

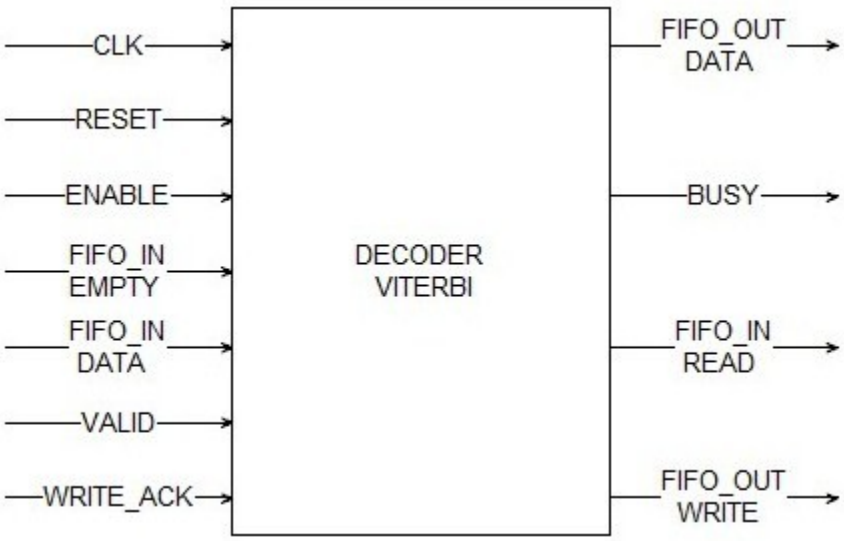

Figura 2.17 - Interfaccia del decoder Viterbi

Il decodificatore viene istanziato come component all'interno del progetto collegandolo opportunamente alla struttura di collaudo. I dati in ingresso verranno prelevati dalla FIFO coded, mentre i risultati della decodifica verranno scritti nella FIFO decoded. Il segnale *enable* andrà collegato nell' *Enable_byte* per permettere l'attivazione del componente da parte del picoBlaze. Inoltre, il segnale *busy* indicherà il funzionamento del decodificatore nel byte *System_status*.

Viene eseguito il test con parole da 8 byte, maschera di errore random, 2048 parole. Si effettua la configurazione del test tramite il programma ViterbiConfig, quindi si esegue il collaudo con configurazione da file, ottenendo i seguenti risultati.

ERRORI PER PAROLA IN INGRESSO	ERRORI DI DECODIFICA	FER %	TEMPO MEDIO CONVERSIONE [ns]
0	0	0,000	3886
1	128	6,250	3877
2	243	11,865	4070
3	355	17,334	4172
4	502	24,512	4445
5	652	31,836	4400
6	800	39,063	4279
7	1008	49,219	4002
8	1203	58,740	3603
9	1342	65,527	3982
10	1564	76,367	3831
MEDIE	779,7	38,071	4454,7

Capitolo 3

Conclusioni e risultati

La procedura di collaudo risulta funzionare correttamente. L'implementazione su FPGA XC3S200 richiede il 14% delle slices e 9 block ram. In caso di necessità, è possibile ridurre l'occupazione di risorse eliminando le unità opzionali e ottenendo l'uso del 11% delle risorse. Non è possibile ridurre il numero di block ram impiegate.

Nelle tabelle seguenti è illustrata l'occupazione di risorse, rispettivamente con e senza le unità opzionali.

Riepilogo Utilizzo del Dispositivo			
Utilizzazione Logica	**Usati**	**Disponibili**	**Uso%**
Numero di Flip Flops delle Slices	286	3,840	7%
Numero di LUT a 4 ingressi	402	3,840	10%
Distribuzione Logica			
Numero di Slices utilizzate	285	1,920	14%
Numero di Slices contenenti solo logica correlata	285	285	100%
Numero di Slices contenenti solo logica non correlata	0	285	0%
Numero Totale LUT a 4 Ingressi	447	3,840	11%
Numero utilizzate per logic	298		
Numero utilizzate per attraversamento	45		
Numero utilizzate per Dual Port RAM	16		
Numero utilizzate per RAM 32x1	52		
Numero utilizzate per Shift registers	36		
Numero di IOB utilizzati	27	173	15%
Numero di RAMB16s	3	12	25%
Numero di BUFGMUXs	1	8	12%

Riepilogo Utilizzo del Dispositivo			
Utilizzazione Logica	**Usati**	**Disponibili**	**Uso%**
Numero di Flip Flops delle Slices	215	3,840	5%
Numero di LUT a 4 ingressi	310	3,840	8%
Distribuzione Logica			
Numero di Slices utilizzate	220	1,920	11%
Numero di Slices contenenti solo logica correlata	220	220	100%
Numero di Slices contenenti solo logica non correlata	0	220	0%
Numero Totale LUT a 4 Ingressi	344	3,840	8%
Numero utilizzate per logic	206		
Numero utilizzate per attraversamento	34		
Numero utilizzate per Dual Port RAM	16		
Numero utilizzate per RAM 32x1	52		
Numero utilizzate per Shift registers	36		
Numero di IOB utilizzati	27	173	15%
Numero di RAMB16s	3	12	25%
Numero di BUFGMUXs	1	8	12%

Al progetto è possibile apportare delle migliorie, alcune prettamente estetiche altre funzionali.

L'interfaccia utente è molto scarna e poco user-friendly oltre a essere testuale: questo perché la procedura di rivolge principalmente a sviluppatori. Nel caso si voglia impiegarla nel test di produzione, una interfaccia grafica è sicuramente auspicabile in quanto migliora l'efficienza del collaudatore.

L'applicazione di supervisione ha subito numerose modifiche dalla prima stesura del codice, portando al proliferare di arrays per la memorizzazione delle informazioni. Una riscrittura del codice in un'ottica più object oriented rispetto la procedurale impiegata può sensibilmente migliorare la comprensibilità del codice oltre a poter portare vantaggi in fase di esecuzione. Ad esempio, si potrebbe definire un oggetto di tipo "dato" il cui costruttore genera la parola casuale, la codifica e genera la maschera di errore casuale (il tutto corredato da opportuni metodi per leggere e modificare i campi dell'oggetto).

Il quantitativo di dati a disposizione del decodificatore è piuttosto limitato e la struttura realizzata impiega le block ram presenti nell'FPGA. Sacrificando delle risorse, è possibile realizzare le memorie FIFO con la RAM installata sulla S3SKB: si potrebbero quindi elaborare circa 500kB di dati al posto di 4kB liberando contestualmente 8 block ram che sarebbero quindi a disposizione del decodificatore.

La comunicazione tramite protocollo RS232c è sicuramente affidabile ma lenta, soprattutto se si prevede di utilizzare grandi quantità di dati; inoltre, l'installazione dell'applicazione di supervisione prevede anche l'installazione di librerie per l'interprete Java. L'utilizzo di altre interfacce di comunicazione, come Ethernet, USB o Firewire ovvierebbe al problema e permetterebbe un maggiore bit rate velocizzando lo scambio di dati tra PC e S3SKB. Inoltre sarebbe possibile valutare l'implementazione di una comunicazione full-duplex con il picoBlaze e quindi la realizzazione di un collaudo in modalità "streaming", liberando per di più le risorse legate alla gestione delle memorie FIFO e le memorie stesse.

Bibliografia

[1] M. Zwolinski; VHDL – Progetto di sistemi digitali. Ed. Pearson Prentice Hall.

[2] D. Vogrig; Dispense del corso “Laboratorio di Elettronica Digitale” Università degli Studi di Padova.

[3] Xilinx; Spartan-3 FPGA Family Data Sheet.

[4] Xilinx; Spartan-3 FPGA Starter Kit Board User Guide.

[5] De Santis, M. Cacciaglia, C. Saggese; Sistemi, automazione e organizzazione della produzione. Ed. Calderini.

[6] C. S. Horstmann; Concetti di informatica e fondamenti di Java 2. Ed. Apogeo.

[7] G. Biondo, E. Sacchi; Manuale di Elettronica e Telecomunicazioni. Ed. Hoepli.

[8] Xilinx; PicoBlaze 8-bit Embedded Microcontroller User Guide.

[9] Xilinx; UART Transmitter and Receiver Macros User Manual.

[10] Xilinx; LogiCORE IP FIFO Generator v4.3 User Guide.

[11] Sun Microsistems Java API http://java.sun.com/reference/index.jsp .

[12] Documentazione Libreria RXTX http://www.rxtx.org .

[13] G. Miorandi; Tecniche di Viterbi per la decodifica su FPGA. Tesi di Laurea Università degli Studi di Padova.

Ringraziamenti

E’ mio desiderio ringraziare tutte le persone che mi hanno permesso di arrivare fin qui: in primis la mia famiglia che sopporta le mie sfuriate e il mio caratteraccio, e in particolare mia nonna Bruna che mi ha sostenuto moralmente nei momenti di difficoltà e culinariamente durante le giornate di lavoro. Un enorme merito lo hanno anche tutti gli amici e colleghi, che hanno avuto la pazienza di ascoltare le mie disquisizioni ingegneristiche: Maurizio “Boodie”, Andrea “Pero”, Manuel “Mago”, Elisa “Puppy”, Elisa “Sbabi”, Matteo “Rustic”, Matteo “Putre”, Marco Mattia “Biondo”, Marco “QG”, Leonardo, Alberto “Maso”, Fabrizio, Carlotta “Lotti”, Nicolò. Una nota particolare a Gabriele, che è riuscito a sopravvivere alle discussioni sulle specifiche del progetto.

E’ inoltre doveroso ringraziare il Prof. Daniele Vogrig che si è preso l’onere di essere il mio relatore.

Non me ne vogliano le persone che non ho nominato: sappiano che io le apprezzo e che nessuno potrà mai negare ciò che sono per me.

Indice delle immagini

Introduzione

Capitolo 1 – Hardware, Software e Interfaccia di Comunicazione

Capitolo 2 – Il progetto