ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SCUOLA DI INGEGNERIA E ARCHITETTURA (SEDE DI BOLOGNA)

CORSO DI LAUREA TRIENNALE
IN
INGEGNERIA INFORMATICA

Streaming di immagini via ethernet con sistemi operativi Linux e Baremetal per un sistema di visione embedded con elaborazione su FPGA

Tesi di laurea sperimentale

CANDIDATO
Mattia Bernasconi

RELATORE
Prof. Stefano Mattoccia

SESSIONE III

Anno Accademico 2015/16

INDICE

Capitolo 1 - Introduzione

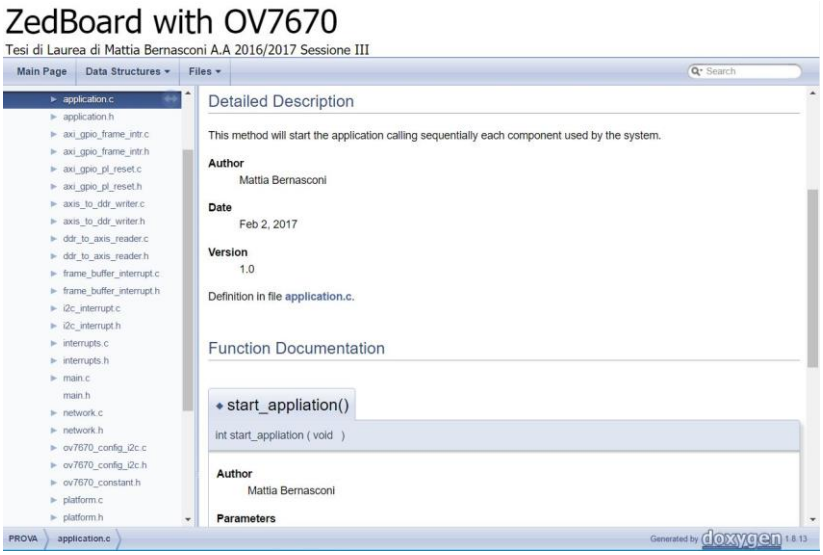
1.1 Obiettivi

Il seguente lavoro di tesi sperimentale è volto a migliorare l'efficienza, la manutenibilità ed estendere le funzionalità di un sistema (già esistente) capace di elaborare e trasmettere immagini ad un cliente remoto acquisite grazie all'utilizzo di sensori di immagine digitali *OV7670*.
Le immagini acquisite da tali sensori vengono salvate all'interno della memoria volatile (RAM) pronte per poter essere eventualmente elaborate e trasmesse ad un dispositivo remoto oppure su un monitor direttamente collegato alla scheda elettronica *ZedBoard* [1].
Obiettivo primario di questa tesi è riorganizzare e ingegnerizzare tramite singoli moduli software il progetto sviluppato nelle precedenti tesi di Riccardo Albertazzi [15] e Simone Mingarelli [16].

I vantaggi della modularizzazione di un software sono molteplici, tra i principali troviamo:

- La possibilità di considerare il modulo come una scatola nera, basterà dunque conoscere i parametri da fornire in ingresso per ottenere in uscita i valori attesi.
- Astrazione dal funzionamento del modulo: non è necessario modificare nulla, né conoscere gli algoritmi interni per il corretto funzionamento del componente, basterà includerlo all'interno del nostro progetto per avere una soluzione già pronta al problema.
- Facilità nel futuro utilizzo del modulo in nuovi progetti: nel caso in cui in un altro progetto si presenti un problema della stessa natura, il nostro componente può essere riutilizzato per velocizzare le fasi di sviluppo.

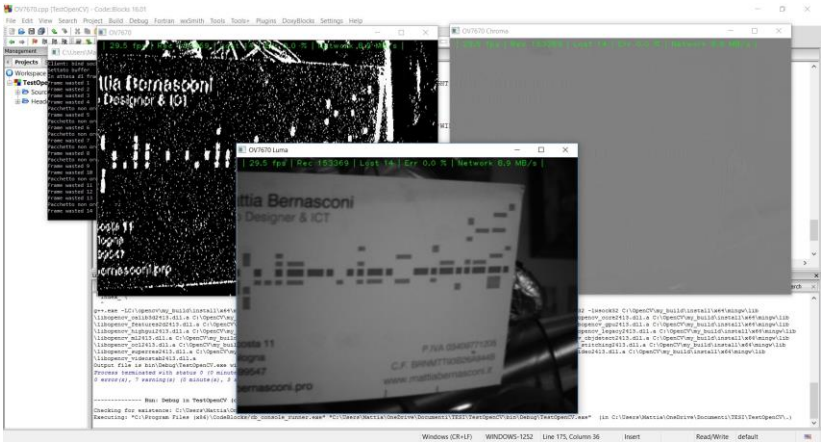
Per permettere una facile lettura del codice e una futura manutenibilità dello stesso, è stata prodotta in formato digitale una documentazione approfondita, come mostrato nella figura seguente, circa l'utilizzo ed il comportamento delle singole funzioni, realizzate durante il processo di ingegnerizzazione del software sviluppato per la board.



1.1 Esempio documentazione generata tramite DoxyGen [3]

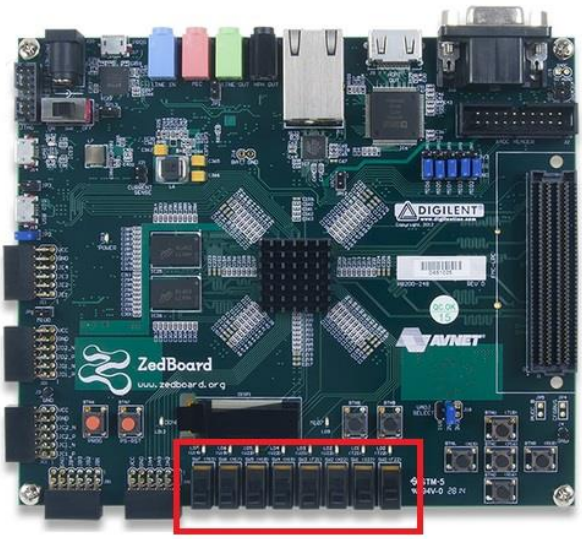
Come accennato in precedenza, un altro aspetto importante trattato in questa tesi è l'estensione delle funzionalità del progetto precedente quali, ad esempio, la trasmissione e ricezione delle immagini, sia in scala di grigio sia nelle componenti di *luminanza* e *crominanza*, e l'inserimento del protocollo di comunicazione sicuro per permettere ad un qualsiasi cliente connesso tramite ethernet alla board di inviare segnali a quest'ultima per poterne modificarne il suo comportamento durante il normale funzionamento.

Prima delle modifiche introdotte nel lavoro svolto, il sistema trasmetteva le immagini acquisite dai sensori *OV7670* in scala di grigi ed eventualmente elaborate scartando le componenti di colore.
La telecamera utilizzata è capace di acquisire immagini nello spazio di colore *YUV*: ogni frame è diviso nelle componenti di luminanza (segnale che trasporta l'informazione relativa alla componente in scala di grigi, grayscale) e crominanza (proprietà dello spettro di colore). Vengono dunque trasmesse tutte e tre le immagini al client remoto che sarà così in grado di ricostruire l'immagine originale a colori per poterla confrontare con quella in scala di grigi elaborata dalla board mediante opportuni algoritmi implementati sulla FPGA (Field Programmable Gate Array) presente nel dispositivo Zynq.



1.2 Esempio ricezione immagini Greyscale (elaborata), Luma e Chroma originali generate dal sensore digitale

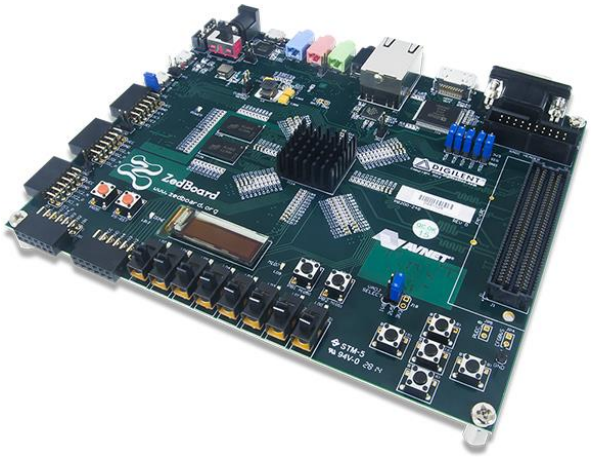
Grazie all'implementazione di un metodo veloce, sicuro ed affidabile quale il protocollo TCP risulta ora possibile modificare a *runtime* i parametri interni alla scheda, permettendo così all'utente di eseguire operazioni preimpostate ed ottenerne immediatamente il risultato, ad esempio l'utilizzatore potrebbe decidere in qualsiasi momento di interrompere la trasmissione delle immagini a colori e mantenere solo quelle in scala di grigi, oppure applicare dei filtri sulle immagini senza bisogno di agire fisicamente sugli interruttori presenti sulla scheda (figura 1.3).



1.3 Dip switch presenti sulla ZedBoard [1]

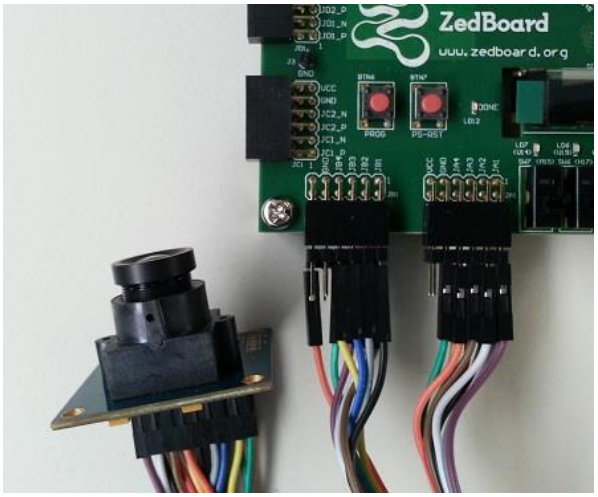
1.2 Contesto Applicativo

Il contesto applicativo di riferimento è quello della computer vision per sistemi embedded, spesso indicato come *Embedded Computer Vision*. Lo strumento utilizzato per le operazioni di sviluppo, debugging e testing dell'intero lavoro svolto è composta da una scheda *"ZedBoard Zynq-7000 ARM/FPGA SoC"* le cui caratteristiche tecniche si prestano allo sviluppo di sistemi innovativi come quelli di questo lavoro.



1.4 Xilinx [2] *ZedBoard*

I sensori utilizzati e connessi direttamente alla scheda sono gli *Omnivision "OV7670"* [10] capaci di acquisire immagini nello spazio di colore *YUV* e di essere configurati tramite il protocollo *I2C*.



1.5 Sensori OV7670 collegati alla ZedBoard

Capitolo 2 - Strumenti Utilizzati

2.1 Ambiente di sviluppo

Per poter portare a compimento nel migliore dei modi e per facilitare la scrittura del codice discusso in fase di analisi, si è reso indispensabile l'utilizzo di alcuni strumenti software.

Per tutta la parte di programmazione sia lato evaluation board sia lato client sono stati utilizzati software proprietari e *open source,* e librerie *cross-platform* per una maggiore portabilità dei sistemi sviluppati su architetture anche molto diverse tra loro.

Il linguaggio di programmazione scelto per lo sviluppo di tutte le componenti del sistema è il C. Lato board il linguaggio C permette una facile gestione dell'hardware e degli interrupt che vengono generati durante il normale funzionamento del sistema. Altro vantaggio derivante da questa scelta è la possibilità di adattare facilmente il codice scritto per il sistema *Bare Metal* a un sistema operativo più completo quale ad esempio Linux nella versione *Petalinux.*

Nell'implementare la parte di trasmissione delle immagini acquisite tramite i sensori *OV7670* attraverso uno *stream* UDP e per la ricezione di comandi tramite protocollo TCP sono state utilizzate librerie di terze parti facilmente utilizzabili ed ottimizzate per sistemi embedded.

Lato client il linguaggio C permette di rendere il software portabile su sistemi operativi sia della famiglia Windows sia della famiglia Unix. Anche in questo caso per astrarre il più possibile il software per la ricezione dei flussi video dal sistema operativo sottostante si è reso indispensabile l'utilizzo di librerie di terze parti. Per poter correttamente utilizzare il software sviluppato basta quindi ricompilare il progetto per il sistema operativo di destinazione senza bisogno di apportare alcuna modifica al codice già scritto.

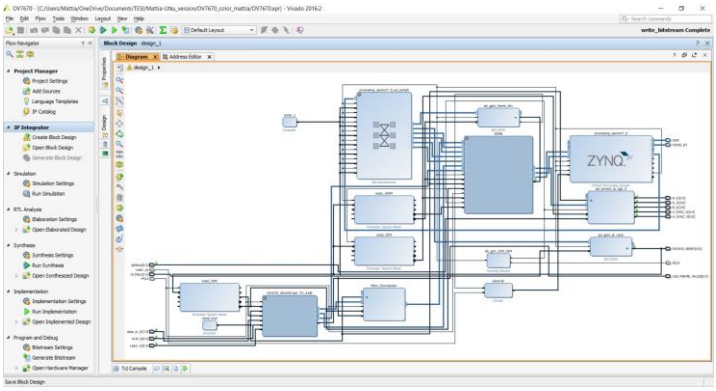
Una volta acquisito il flusso dati video trasmesso in broadcast tramite protocollo UDP dalla *ZedBoard*, questo viene elaborato e visualizzato sotto forma di immagini grazie all'utilizzo delle librerie *OpenCV* [4]. Tra i vantaggi derivanti dall'utilizzo di queste librerie due degli aspetti più apprezzati sono:

- La possibilità di inserire facilmente dati in sovrimpressione alle immagini mostrate a video.
- La possibilità di ricreare uno stream video nello spazio di colore RGB dai flussi di immagini *luma* e *chroma*.
- La possibilità di eseguire algoritmi di computer vision, anche molto complessi, per varie finalità applicative.

2.1.1 Vivado e Vivado SDK

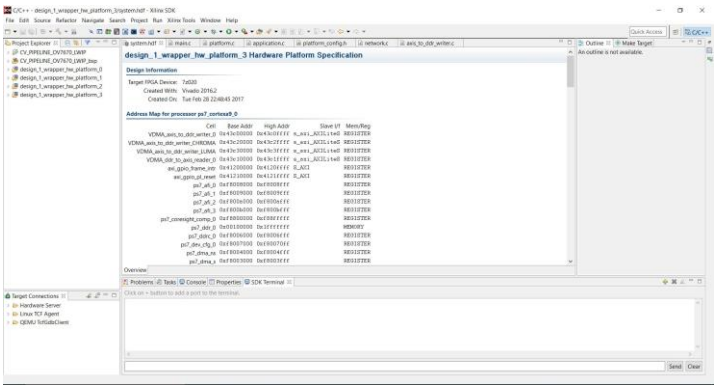
Per poter correttamente programmare la *ZedBoard* sia lato hardware che lato software è stato necessario utilizzare due ambienti di sviluppo (IDE). Nella fattispecie sono stati utilizzati *Xilinx Vivado* [5] e *Xilinx Vivado SDK* [6].

Un particolare strumento di Vivado, denominato *Vivado HLS* consente la creazione e la verifica dei blocchi di IP core direttamente a partire da specifiche algoritmiche in C, producendo in tempi rapidi progetti che eguagliano in prestazioni il codice RTL scritto manualmente, con una fase di verifica di molto più veloce rispetto alla simulazione RTL. Vivado supporta anche un ecosistema in continua crescita di librerie software realizzabili su hardware.



2.1 Block Design creato con Xilinx Vivado

Xilinx Vivado *SDK* [6] (versione modificata da Xilinx del noto IDE *Eclipse* [7]) include la strumentazione operativa integrata nel sistema e la visualizzazione delle prestazioni per trovare rapidamente i colli di bottiglia delle prestazioni a livello di sistema e per far girare scenari possibili su condizione. Xilinx SDK fornisce dei generatori di traffico AXI configurabili che operano sulla FPGA per rendere possibile lo sviluppo del software dedicato fin dalle prime fasi di sviluppo.



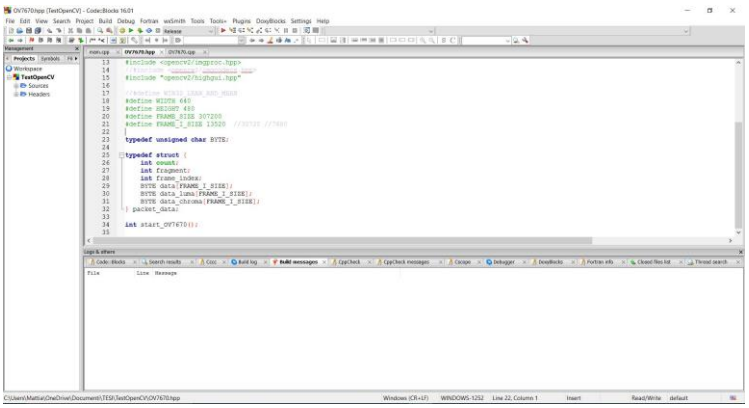
2.2 Design Information di Xilinx[2] Vivado SDK [2]

2.1.2 CodeBlocks e OpenCV

Per poter visualizzare il flusso di immagini ricevute dalla board sotto forma di stream video, si è scelto di utilizzare le librerie open source *OpenCV [4]* tramite l'ambiente di sviluppo (IDE) *CodeBlocks* [8] con compilatore (principalmente utilizzato in ambiente Windows) *MinGW* [9].

Per facilitare altri utenti nella futura installazione dell'ambiente di sviluppo in ambiente Windows necessario all'aggiornamento del client tramite *CodeBlocks + OpenCV + MinGW* è stata redatta una guida illustrata in formato *pdf* che rimarrà allegata a questa tesi (supporto elettronico).

CodeBlocks è un IDE libero, open source e multipiattaforma. È scritto in C++ usando wxWidgets. Grazie ad un'architettura basata su plugin, le sue capacità e caratteristiche sono estese proprio dai plugin installati. Caratteristica di particolare interesse è la possibilità di installare questo IDE su quasi tutti i principali sistemi operativi oggi disponibili come Linux, Windows e MacOS.



2.3 Anteprima di CodeBlocks configurato correttamente

OpenCV (acronimo in lingua inglese di Open Source Computer Vision Library) è una libreria software multipiattaforma ampiamente utilizzata nell'ambito della visione artificiale.

È un software libero originariamente sviluppato da Intel, centro di ricerca in Russia di Nižnij Novgorod. Successivamente fu poi mantenuto da Willow Garage e ora da Itseez.

Il linguaggio di programmazione principalmente utilizzato per sviluppare queste librerie è il C++, è possibile utilizzare queste librerie anche attraverso altri linguaggi, quali ad esempio il C, il Python e Java.

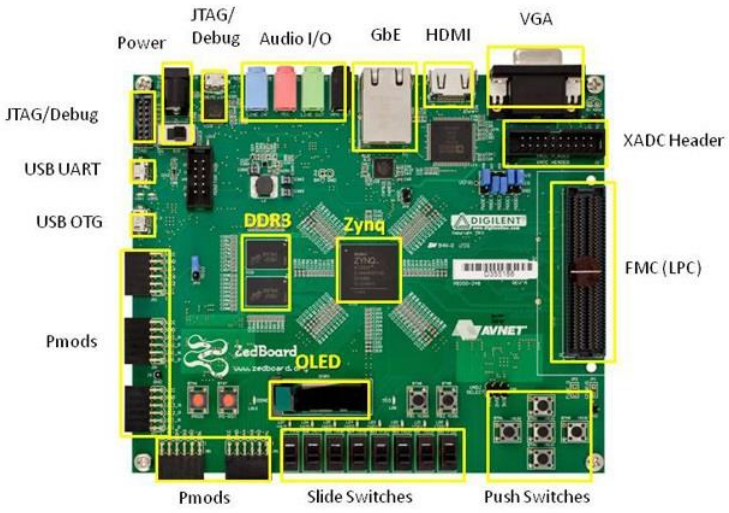
2.2 ZedBoard

ZedBoard è una scheda di sviluppo dai costi contenuti per il SoC Zynq-7000 di Xilinx. La scheda contiene tutto il necessario per creare una configurazione basata su *Linux*, *Android*, *Windows* o altro progetto SO/RTOS. Diversi connettori d'espansione rendono visibili I/O logici programmabili e il sistema di elaborazione per il facile accesso da parte dell'utente. Grande vantaggio del sistema di elaborazione ARM, accoppiato al SoC Zynq-7000 e dalla logica programmabile della serie 7, è il poter creare progetti unici e potenti. *ZedBoard* è supportato dal sito web della comunità *zedboard.org* dove gli utenti possono collaborare con altri ingegneri o appassionati che lavorano su progetti basati su Zynq.

Sintesi delle porte di I/O e caratteristiche tecniche della board:

- XC7Z020-CLG484-1 SoC Zynq-7000
- Memoria SDRAM DDR3 da 512 MB
- 256 Mbit di memoria Flash seriale Quad-SPI
- Display OLED 128 x 32
- CODEC audio I²S
- Connettore femmina scheda SD
- Programmazione USB-JTAG a bordo scheda
- Connettore femmina RJ45: Ethernet 10/100/1000
- USB OTG 2.0 e ponte USB-UART
- Espansione I/O PS e PL (FMC, compatibile con Pmod™, XADC)
- Connettore femmina HDMI 1080p
- Connettore femmina VGA

Applicazioni principali della ZedBoard sono elaborazione video, controllo di motori, accelerazione software, sviluppo Linux/Android/RTOS, elaborazione ARM incorporata, prototipazione generale SoC Zynq-7000.



2.4 Posizione delle porte e dei componenti della ZedBoard

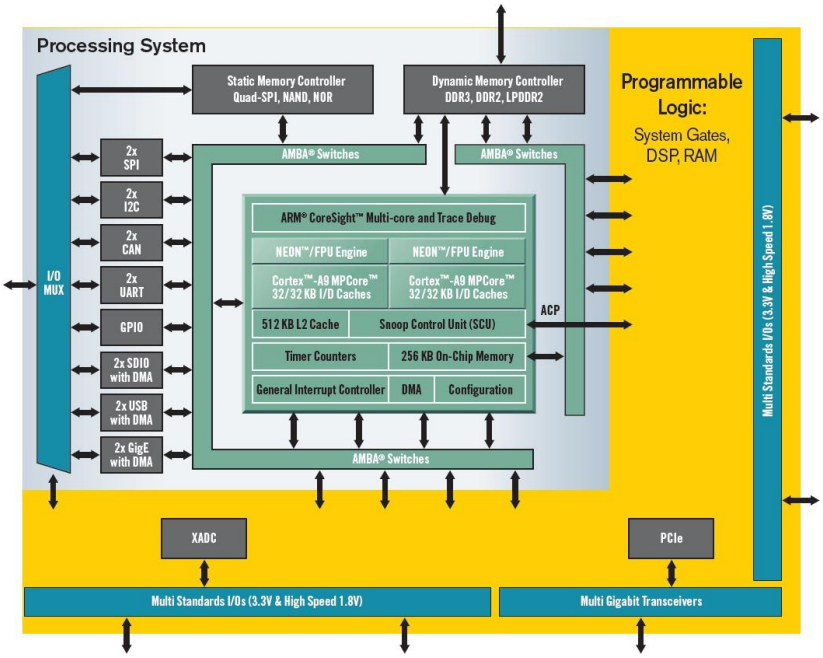
In elettronica digitale, un dispositivo FPGA, è un circuito integrato le cui funzionalità sono programmabili via software. Tali dispositivi consentono l'implementazione di funzioni logiche anche molto complesse, e sono caratterizzati da un'elevata scalabilità. Questo tipo di tecnologia ha assunto un ruolo sempre più importante nell'elettronica industriale così come nella ricerca scientifica.

Grazie al continuo progredire delle tecniche di miniaturizzazione, le capacità di tali dispositivi sono aumentate enormemente nel corso di due soli decenni, durante i quali si è passati da poche migliaia di porte logiche a qualche milione di porte logiche per singolo dispositivo FPGA.

Questo dispositivo a semiconduttore è costituito da una matrice di blocchi logici configurabili (CLB) collegati tramite interconnessioni programmabili. L'utente stabilisce le interconnessioni attraverso la programmazione SRAM. Un CLB può essere semplice (porte AND, OR, ecc.) o complesso (un blocco di RAM). L'FPGA consente di effettuare modifiche a un progetto anche una volta che il dispositivo è stato saldato in un circuito stampato.

2.2.1 Zynq 7000

I dispositivi delle famiglie di prodotti Zynq-7000 Xilinx e All Programmable SoC (AP SoC) Altera contengono un processore ARM Cortex-A9 *dual core* abbinato a una potente struttura logica programmabile. Tale combinazione consente a ingegneri e scienziati di progettare e implementare algoritmi su un singolo chip che consuma meno spazio ed energia.



2.5 Schematico Zynq-7000 Extensible Processing Platform (EPP)

Questi dispositivi integrano la programmabilità software di un processore basato su ARM con la programmabilità hardware di un FPGA, permettendo analisi chiave e accelerazione hardware, integrando al contempo CPU, DSP, ASSP e funzionalità di segnale misto su un singolo dispositivo. Gli AP SoC Zynq-7000 aggiungono intelligenza personalizzabile ai sistemi embedded odierni per soddisfare le specifiche esigenze di applicazione.

2.2.2 Sistemi Operativi

In computer science per *bare machine* o *bare metal*, si intende un computer senza il suo sistema operativo nativo. I sistemi operativi moderni si sono evoluti attraverso varie fasi, dai sistemi elementari agli odierni sistemi complessi altamente sensibili.

Dopo lo sviluppo di logiche programmabili (non è quindi richiesto un cambiamento fisico della configurazione per ottenere risultati anche molto diversi tra loro), ma prima dello sviluppo di sistemi operativi, i software erano sviluppati direttamente utilizzando linguaggio macchina dai programmatori senza nessun altro supporto. Questo approccio è chiamato "*bare machine*" nello sviluppo di sistemi operativi. Oggi è in gran parte applicabile allo sviluppo di sistemi embedded e firmware, mentre i programmi di uso comune sono gestiti da un *runtime-system* all'interno del sistema operativo.

Alternativa di alto livello al sistema sopra descritto è *Linux*. Questo sistema operativo, per esempio nella versione PetaLinux, ma è possibile installarne anche altre versioni sulla evaluation board, è stato integrato e testato per i dispositivi Xilinx. La distribuzione di riferimento Linux include sia i pacchetti binari sia i sorgenti di Linux, tra cui:

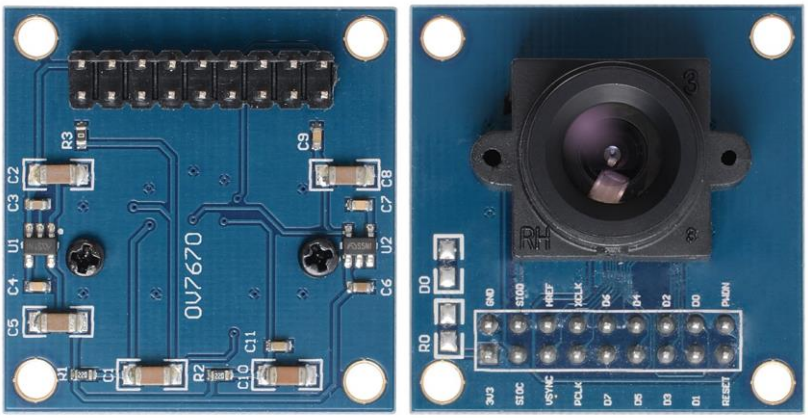
- Boot loader
- Kernel ottimizzato per la CPU
- Applicazioni Linux e librerie
- Strumenti di sviluppo per applicativi C e C ++
- Strumenti di debug
- Supporto Thread ed FPU
- Web server integrato per una facile gestione remota delle configurazioni di rete e del firmware

Capitolo 3 - Architettura del sistema

3.1 Sensori OV7670

La telecamera *OV7670* è un sensore di immagine CMOS a bassa tensione in grado di fornire immagini a risoluzione VGA (640x480), e anche a risoluzioni inferiori, a 30 Hz occupando uno spazio davvero contenuto.

I sensori *OV7670* possono produrre immagini a piena risoluzione, *sub-campionate* in diversi formati, controllati tramite la porta seriale *I2C* (denominata da Omnivision "Serial Camera Control Bus" (SCCB)). Questi sensori sono in grado di catturare e trasmettere fino a 30 fotogrammi al secondo in risoluzione VGA permettendo all'utente di controllare la qualità delle immagini, il trasferimento ed il formato di uscita delle immagini acquisite.



3.1 Sensore OmniVision OV7670 [10] *retro e fronte*

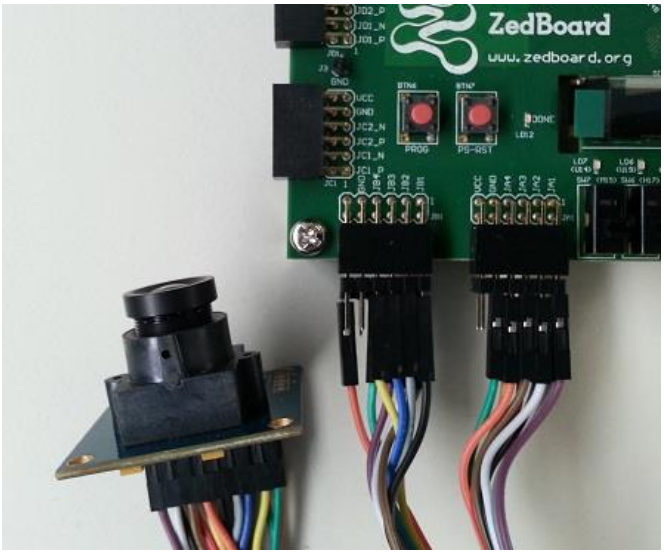
Tutte le funzioni di elaborazione delle immagini richieste, tra cui il controllo dell'esposizione, il bilanciamento del bianco, la saturazione e la gamma sono quindi programmabili attraverso l'interfaccia I2C. Grazie ad una tecnologia proprietaria del produttore OmniVision, la fotocamera è in grado di migliorare la qualità dell'immagine riducendo o eliminando i comuni disturbi derivanti dalle sorgenti luminose artificiali, diminuire il rumore, eliminare sbavature e correggere i principali disturbi derivanti dalla conversione analogico-digitale.

3.1.2 Protocollo I2C

I sensori OV7670 sono collegati alla ZedBoard tramitre tre fasci di cavi, uno di questi è quello che si occupa delle comunicazioni tramite protocollo I2C. Grazie all'utilizzo di questo protocollo è possibile programmare i sensori affinché le immagini trasmesse siano elaborate ancora prima di arrivare alla memoria della board.

Il protocollo I2C (o *IIC*) abbreviazione di Inter Integrated Circuit è un sistema di comunicazione seriale bifilare utilizzato tra circuiti integrati, è composto solitamente da almeno un master e uno o più slave. La situazione più frequente vede un singolo master e più slave; possono tuttavia essere usate architetture multimaster e multislave in sistemi più complessi.

Il bus è stato sviluppato dalla Philips nel 1982 e dopo la realizzazione di centinaia di componenti e sistemi negli anni '80, nel 1992 è stata prodotta la prima versione del protocollo che ha subìto diversi aggiornamenti ed ha generato bus simili. Trattandosi di un protocollo seriale i vantaggi che offre sono quelli di impegnare solo due linee (e quindi due pin dei dispositivi che lo usano).



3.2 Sensori OV7670 collegati alla ZedBoard e configurati tramite I2C

3.2 Interfaccia ethernet 10/100/1000

Per la comunicazione tra board e client remoto si è scelto di utilizzare l'interfaccia di rete ethernet implementando la trasmissione delle informazioni tramite due dei più diffusi protocolli: *UDP* e *TCP*. La ZedBoard dispone di un'interfaccia di rete capace di trasmettere fino ad una velocità massima Gigabit ovvero 1.000 Mbit/s. Per lo streaming video delle immagini acquisite tramite i sensori OV7670 quest'interfaccia si è dimostrata pratica e veloce permettendo così un futuro sviluppo del sistema e consentendo un'estensione delle funzionalità in quanto la banda al momento utilizzata non satura il canale.

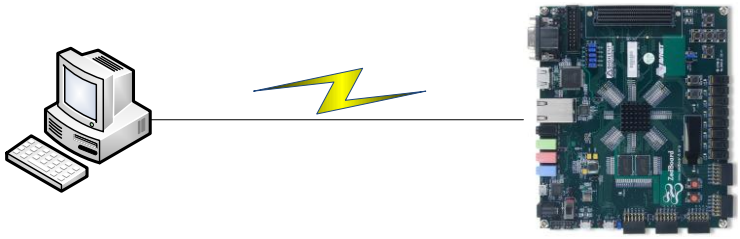
Altro aspetto importante della comunicazione tramite interfaccia ethernet è la possibilità di utilizzare il protocollo *UDP* per la trasmissione in broadcast a tutti i dispositivi fisicamente connessi alla stessa rete (e aventi stessa gamma di indirizzi *IPv4*). Per la trasmissione del flusso video è stato scelto questo protocollo a datagrammi in quanto i frame che vengono mandati con ogni pacchetto non possono essere ritrasmessi in caso di problemi di comunicazione.

Il protocollo *TCP* è stato invece utilizzato per la comunicazione sicura tra board e client remoto, ogni volta che l'utente decide di cambiare a *runtime* il comportamento della scheda si rende indispensabile che l'invio dell'informazione sia affidabile e con una conferma di ricezione dello stesso. Questo protocollo nasce proprio con lo scopo di garantire sicurezza ed affidabilità e per sua natura in caso di errori di comunicazione la trasmissione dei dati viene ripetuta fin quando non si ha la certezza che l'informazione sia arrivata all'altro *endpoint* in maniera corretta.

3.2.1 Connessione punto-punto e rete LAN

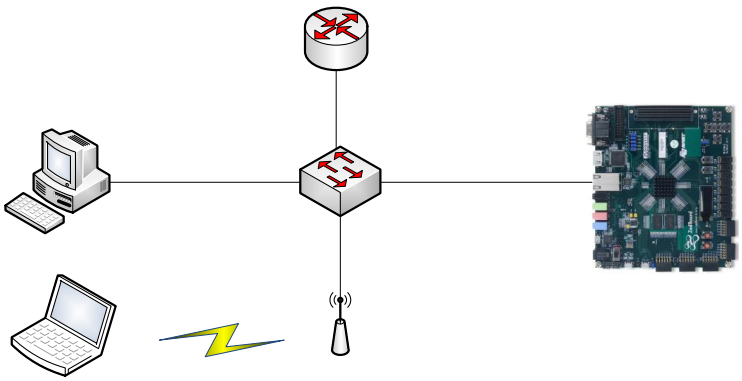
La connessione tramite interfaccia di rete ethernet della ZedBoard a un dispositivo remoto può avvenire in due differenti modalità:

1. Tramite connessione diretta punto-punto, ovvero la board e l'host sono connessi direttamente l'uno con l'altro mediante l'utilizzo di un cavo di rete *UTP cat. 5e o 6*. Questa configurazione presenta il vantaggio di avere una connessione (fisica) sicura e affidabile, i pacchetti inviati dai due dispositivi arrivano con un'altissima probabilità al destinatario in quanto non vi sono periferiche intermedie che si occupano di smistare i pacchetti.



3.3 Diagramma connessione punto-punto tra ZedBoard e Computer

2. Collegando invece sia la board che i vari client ad un intermediario (*hub*, *switch*, *router*, *access point*, ecc.) tutti gli ascoltatori saranno in grado di visualizzare il flusso dati video trasmesso in broadcast sul segmento di rete che il server *DHCP* presente nella rete ha assegnato ai dispositivi ad essa connessi. Per il corretto funzionamento in questa seconda configurazione sarà però necessario impostare sulla ZedBoard l'acquisizione dell'indirizzo *IP* in maniera dinamica e non più impostato staticamente.



3.4 Diagramma connessione LAN tra ZedBoard e n Client

Capitolo 4 - Organizzazione Progetto

4.1 Lato board

Per riorganizzare al meglio l'esistente progetto sviluppato nelle tesi di Riccardo Albertazzi [16] e Simone Mingarelli [17] è stato necessario studiare a fondo ed eseguire un importante lavoro di analisi del codice per capire l'esatta sequenza di invocazione delle funzioni e il punto esatto in cui inizializzare tutti i componenti usati per la corretta elaborazione e trasmissione delle immagini.

Per la reingegnerizzazione e la reimplementazione dell'esistente sistema è stato necessario partire dalla sua progettazione, al fine di migliorarne e aggiungervi funzionalità (ad esempio la ricezione delle immagini a colori e l'introduzione del protocollo TCP), fino ad arrivare alla lettura del codice esistente per cogliere eventuali criticità che avrebbero potuto compromettere l'efficienza e la stabilità del sistema.

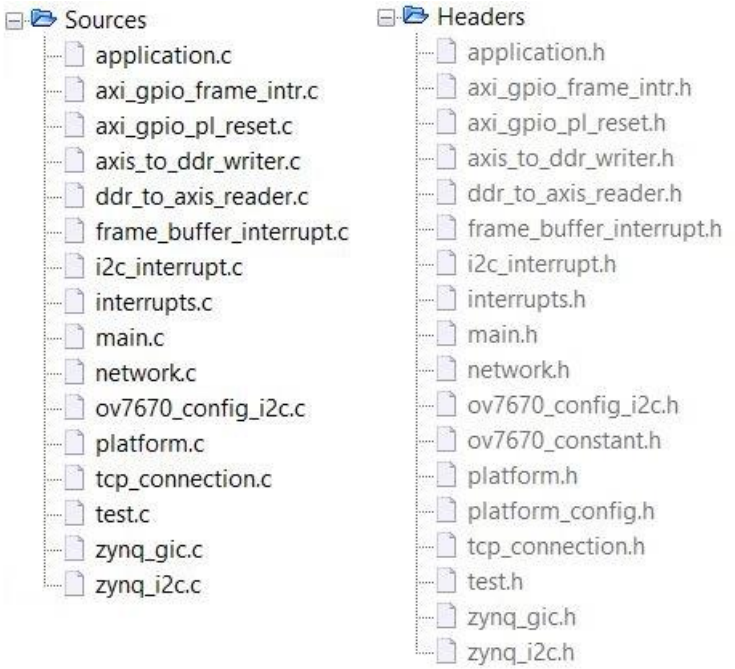
Per rendere inoltre più manutenibile il sistema sono state create moltissime funzioni ognuna contenente poche istruzioni: così facendo risulta più facile ed immediato trovare il modulo che si occupa di gestire un dato componente (ad esempio inizializzare il *writer* o il *reader* dalla memoria *RAM*). Tutte le funzioni inerenti uno stesso componente sono state raggruppate in un unico file diviso in *source* e *header* per consentire astrazione in futuro, basterà infatti conoscere la firma della funzione presente all'interno del file di intestazione per poter utilizzare o sospendere il componente.

All'interno dei file *headers* sono state inserite anche tutte le variabili di visibilità globale e i files da includere necessari al *linker* per poter compilare correttamente il programma. Così facendo, ad esempio per la parte relativa al networking, si può facilmente esportare il file contenete le costanti utilizzate per la trasmissione delle immagini ed importarlo nel progetto di ricezione senza bisogno di scrivere codice ridondato con possibile inserimento di errori.

Tutto il codice scritto è stato commentato in lingua inglese (per una completa comprensione anche da parte di utenti internazionali) sia in maniera "classica" sia con speciali annotazioni che hanno permesso di generare una documentazione specifica sul completo funzionamento del sistema.

4.1.1 Struttura

Come anticipato nel capitolo precedente, il progetto è stato suddiviso in vari file per permettere una facile lettura e comprensione del codice ai futuri sviluppatori del progetto. A ogni file sorgente corrisponde un file di intestazione in cui sono dichiarate le firme delle funzioni utilizzabili dall'esterno. Le strutture e le funzioni di appoggio sono state invece definite solo all'interno del file di origine per evitare di confondere i programmatori e per rendere ogni coppia *sorgente/intestazione* utilizzabile come una scatola nera.



4.1 Files del progetto divisi in Sources e Headers

Come si può notare della figura 4.1 esistono due file di intestazione che non hanno un corrispettivo file di origine. I due files in questione sono *"ov7670_constant.h"* e *"platform_config.h"*. Questi due files contengono tutte le informazioni circa i riferimenti alle aree di memoria *RAM* in cui possiamo trovare i frame catturati e salvati dai sensori *OV7670* e le informazioni circa le porte e le configurazioni per l'utilizzo delle trasmissioni via ethernet.

4.1.1.1 File "ov7670_constant.h"

All'interno del file *"ov7670_constant.h"* sono definite tutte le costanti che il sistema dovrà utilizzare per poter configurare correttamente il gruppo sensori e per poter acquisire da essi i frame delle immagini e salvarle in memoria.

In questo file sono inoltre definiti i riferimenti ai registri e le configurazioni per la sincronizzazione orizzontale e verticale delle immagini acquisite dalla telecamera.

```
 * ov7670_constant.h

#ifndef SRC_OV7670_CONSTANT_H_
#define SRC_OV7670_CONSTANT_H_

#define DATA_REGISTER_CHANGED 1
#define DATA_REGISTER_NOT_CHANGED 0

#define INPUT_CLOCK_FREQUENCY 24//MHz
#define INTERNAL_CLOCK_FREQUENCY 24 //MHz

#define OV7670_SLAVE_ADDRESS 0x21
#define OV7670_SLAVE_ADDRESS_WRITE 0x42
#define OV7670_SLAVE_ADDRESS_READ 0x43

#define AEW 0x24
#define AEW_VALUE 0x95

#define AEB 0x25
#define AEB_VALUE 0x33

#define VPT 0x26
#define VPT_VALUE 0xE3

#define HAECC77 0xAA
#define HAECC77_VALUE_HISTOGRAM_AEC_ON 0x94
#define HAECC77_VALUE_AVERAGE_AEC_ON 0x00


#define CLKRC 0x11
#define CLKRC_VALUE_VGA 0x01
#define CLKRC_VALUE_NIGHTMODE_AUTO 0x80

#define COM7 0x12
#define COM7_VALUE_VGA 0x01
#define COM7_VALUE_VGA_COLOR_BAR 0x03
#define COM7_VALUE_QVGA 0x00
#define COM7_VALUE_RESET 0x80

#define COM3 0x0C
#define COM3_VALUE_VGA 0x00
#define COM3_VALUE_QVGA 0x04
```

```
#define COM13_VYUY 0x89

#define COM16 0x41
#define COM16_VALUE_DENOISE_ON_EDGE_ENHANCEMENT_ON 0x38

#define SATCTR 0xC9
#define SATCTR_VALUE 0x60 //default 0xC0

#define HSTART 0x17
#define HSTART_VALUE_DEFAULT 0x11
#define HSTART_VALUE_VGA 0x13

#define HSTOP 0x18
#define HSTOP_VALUE_DEFAULT 0x61
#define HSTOP_VALUE_VGA 0x01

#define HREF 0x32
#define HREF_VALUE_DEFAULT 0x80
#define HREF_VALUE_VGA 0xB6

#define VSTRT 0x19
#define VSTRT_VALUE_DEFAULT 0x03
#define VSTRT_VALUE_VGA 0x02

#define VSTOP 0x1A
#define VSTOP_VALUE_DEFAULT 0x7B
#define VSTOP_VALUE_VGA 0x7A

#define VREF 0x03
#define VREF_VALUE_DEFAULT 0x03
#define VREF_VALUE_VGA 0x0A

#define ABLC1 0xB1
#define ABLC1_VALUE 0x0C

#define THL_ST 0xB3
#define THL_ST_VALUE 0x82

//#define COM8 0x13
//#define COM8_

#endif /* SRC_OV7670_CONSTANT_H_ */
```

4.2 Parte del contenuto del file "ov7670_costant.h"

4.1.1.2 File "platform_config.h"

All'interno del file *"platfrom_config.h"* sono definite tutte le costanti e le strutture dati che il sistema dovrà utilizzare per poter trasmettere le immagini.

All'interno di questo file di intestazione vengono inoltre definite le costanti riguardanti la dimensione di ogni frammento dei singoli pacchetti *UDP* calcolate grazie all'utilizzo di *WireShark* [11] (software per l'analisi approfondita della rete) in maniera tale da rendere la perdita dei pacchetti il più possibile prossima allo zero.

In questo file viene anche definita la struttura *"packet_data"* che contiene i contatori circa il numero del frammento e del frame e le informazioni sotto forma di array di byte delle tre immagini trasmesse:

- *"data"* per le immagini *grayscale* elaborate dalla board su FPGA
- *"data_luma"* per le immagini contenenti le informazioni sulla luminanza
- *"data_chroma"* per le immagini contenenti le informazioni sulla crominanza

```
#define __PLATFORM_CONFIG_H_

#define STDOUT_IS_PS7_UART
#define UART_DEVICE_ID 0

#include <stdio.h>
#include "xaxis_to_ddr_writer.h"
#include "xddr_to_axis_reader.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xiicps.h"

#define MTU_SIZE 1500
#define FRAME_SIZE 307200
#define FRAME_UDP_FRAGMENT_SIZE 13520 //30720 //7680

#define LOCAL_PORT 5454
#define REMOTE_PORT 5555
#define PLATFORM_EMAC_BASEADDR XPAR_XEMACPS_0_BASEADDR

#define FRAME_BUFFER_DIM 307200
#define FRAME_BUFFER_BASE_ADDR 0x10000000
#define FRAME_BUFFER_NUM 8

#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID

#define RESET_ENABLED 0
#define RESET_DISABLED 1

typedef unsigned char BYTE;

typedef struct{
    int count;
    int fragment;
    int frame_index;
    BYTE data[FRAME_UDP_FRAGMENT_SIZE];
    BYTE data_luma[FRAME_UDP_FRAGMENT_SIZE];
    BYTE data_chroma[FRAME_UDP_FRAGMENT_SIZE];
}packet_data;

int frame_index;

#endif
```

4.3 Contenuto del file "platform_confg.h"

4.1.1.3 File "main.c"

Come noto a tutti i programmatori, il *main* è il file principale dell'applicazione ed il primo che viene eseguito una volta lanciato il programma. Essendo questo lavoro di tesi volto alla modularizzazione del sistema già esistente, questo file risulta essere molto differente dalla versione esistente in quanto si occupa semplicemente di richiamare le due funzioni principali dell'applicativo:

1. *"init_platform"* che si occuperà di inizializzare tutti i componenti e le variabili utilizzate dal sistema.
2. *"start_application"* che si occuperà di configurare e lanciare in esecuzione i thread e le applicazioni *UDP* e *TCP*.

```
/*
 * Created on: Feb 2, 2017
 *     Author: Mattia Bernasconi
 *       Mail: mattia@studiobernasconi.com
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ----------------------------------------------
 * | UART TYPE   BAUD RATE                      |
 * ----------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include "main.h"

int main()
{
    int result;

    //Platform Initialization
    result = init_platform();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about init_platform\n");
    }
    xil_printf("init_platform done \n\r");

    //Application Starting
    result = start_appliation();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about start_appliation\n");
    }
    xil_printf("Application started\n");

    //cleanup_platform();

    return 0;
}
```

4.4 Contenuto del file "main.c"

4.1.1.4 File "platform.c"

Questo file contiene un'unica funzione che ha il compito di inizializzare tutti i componenti legati alla piattaforma. Ogni chiamata verifica che il processo di inizializzazione sia andato a buon fine, in caso di successo o errore viene stampato a video (visualizzabile tramite monitor seriale) l'esito della procedura.

Nello specifico sono inizializzati i *reader* e *writer* per la lettura e la scrittura nella memoria volatile *RAM*, gli *interrupt* annidati, il *GIC* (General Interrupt Controller) dell'ARM e infine la scheda di rete *ethernet*.

```
int init_platform(void)
{
    xil_printf("[INFO] Starting ARM application\n");
    int result;

    //DDR Writer
    result = init_axis_to_ddr_writer();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about axis_to_ddr_writer\n\r");
    }
    xil_printf("axis_to_ddr_writer done \n\r");

    //DDR Writer LUMA
    result = init_axis_to_ddr_writer_luma();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about axis_to_ddr_writer\n\r");
    }
    xil_printf("axis_to_ddr_writer done \n\r");

    //DDR Writer CHROMA
    result = init_axis_to_ddr_writer_chroma();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about axis_to_ddr_writer\n\r");
    }
    xil_printf("axis_to_ddr_writer done \n\r");

    //DDR Reader
    result = init_ddr_to_axis_reader();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about ddr_to_axis_reader\n\r");
    }
    xil_printf("ddr_to_axis_reader done \n\r");

    //GPIO Frame Intr
    result = init_axi_gpio_frame_intr();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about axi_gpio_frame_intr\n\r");
    }
    xil_printf("axi_gpio_frame_intr done \n\r");

    //GPIO PL Reset
    result = init_axi_gpio_pl_reset();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about axi_gpio_pl_reset\n\r");
    }
    xil_printf("axi_gpio_pl_reset done \n\r");

    //ZYNQ I2C
    result = init_zynq_i2c(IIC_DEVICE_ID, &Iic, &Config);
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about zynq_i2c\n\r");
    }
    xil_printf("zynq_i2c done \n\r");

    //ZYNQ GIC
    result = init_zynq_gic();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about init_zynq_gic\n\r");
    }
    xil_printf("init_zynq_gic done \n\r");

    //INIT UDP
    result = init_network();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about init_network\n\r");
    }
    xil_printf("init_network done \n\r");

    return XST_SUCCESS;
}
```

4.5 Contenuto del file "platfom.c"

4.1.1.5 File "application.c"

Una volta completata l'inizializzazione di tutte le periferiche, strutture dati e variabili necessarie al funzionamento del sistema, è necessario eseguire le configurazioni dei componenti ed eseguire le applicazioni.

Prima di compiere le configurazioni e lanciare gli applicativi è eseguito un "*soft reset*", successivamente vengono configurate le priorità degli *interrupts*, i *reader* e *writer* per la *DDR*, l'applicazione *UDP* viene messa in esecuzione, vengono abilitati gli *interrupts*, il gruppo sensori *OV7670* viene configurato tramite protocollo *IIC* ed infine anche l'applicazione *TCP* viene lanciata.

Solo per scopo di testing (si può notare dal commento) prima di lanciare l'applicativo *TCP* è eseguito uno *stress-test* per verificare la corretta configurazione degli *interrupts* e che la lettura tramite *I2C* funzioni correttamente.

```
int start_appliation()
{
    int result;

    //RESET PL
    result = reset_PL();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about reset_PL\n\r");
    }
    xil_printf("reset_PL done \n\r");

    //CONFIGURE INTERRUPTS
    result = configure_interrupts();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configure_interrupts\n\r");
    }
    xil_printf("configure_interrupts done \n\r");

    //DISABLE INTERRUPTS
    result = disable_interrupts();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about disable_interrupts\n\r");
    }
    xil_printf("disable_interrupts done \n\r");

    //CONFIGURE DDR WRITER
    result = configue_axis_to_ddr_writer();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configue_axis_to_ddr_writer\n\r");
    }
    xil_printf("configue_axis_to_ddr_writer done \n\r");

    //CONFIGURE DDR WRITER LUMA
    result = configue_axis_to_ddr_writer_luma();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configue_axis_to_ddr_writer\n\r");
    }
    xil_printf("configue_axis_to_ddr_writer done \n\r");

    //CONFIGURE DDR WRITER CHROMA
    result = configue_axis_to_ddr_writer_chroma();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configue_axis_to_ddr_writer\n\r");
    }
    xil_printf("configue_axis_to_ddr_writer done \n\r");

    //CONFIGURE DDR READER
    result = configue_ddr_to_axis_reader();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configue_ddr_to_axis_reader\n\r");
    }
    xil_printf("configue_ddr_to_axis_reader done \n\r");

    //START UDP APPLICATION
    result = start_udp();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about start_udp\n\r");
    }
    xil_printf("start_udp done \n\r");

    //ENABLE INTERRUPTS
    result = enable_interrupts();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about enable_interrupts\n\r");
    }
    xil_printf("enable_interrupts done \n\r");

    //CONFIGURE CAMERA BY I2C
    result = OV7670_configure();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configue_camera_by_i2c\n\r");
    }
    xil_printf("configue_camera_by_i2c done \n\r");

    //TEST NEASTED INTERRUPT
    //Remove comment to test IIC read communication
    //test_neasted_interrupts();

    //START TCP APPLICATION
    result = start_tcp();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about start_tcp\n\r");
    }
    xil_printf("start_tcp done \n\r");

    return XST_SUCCESS;
}
```

4.6 Contenuto del file "application.c"

4.1.1.6 File “test.c”

Questo file ha il solo scopo di fornire funzioni per stressare il sistema e verificare che anche in situazioni critiche tutto continui a funzionare correttamente.

Essendo un file contenete funzioni di testing, queste sono lanciate in esecuzione solamente sotto controllo del programmatore e mai durante una normale esecuzione del progetto completo. Nella figura 4.7 è riportato l’inizio del file in cui si può notare come la funzione “test_neasted_interrupts” esegua uno *stress-test* per verificare che gli *interrupts* siano correttamente annidati e che leggendo 100 volte nel registro 0x0A e 100 volte nel registro 0x0B dei sensori *OV7670* tutto funzioni alla perfezione.

```
/*
 * test.c
 *
 *  Created on: Feb 13, 2017
 *      Author: Mattia Bernasconi
 *        Mail: mattia@studiobernasconi.com
 */
#include "test.h"

//Test per interrupt annidati
void test_neasted_interrupts()
{
    //NUMBER OF TEST LOOP
    static int MAX_TEST = 100;

    BYTE slave_address, slave_register, read_data_register;
    slave_address = OV7670_SLAVE_ADDRESS;

    xil_printf("\n***TESTING IIC READ AT 0x0A FOR %d TIMES***\n", MAX_TEST);
    slave_register = 0x0A;
    int i;
    for(i = 1; i <= MAX_TEST; i++)
    {
        I2C_read_8_bit(slave_address, slave_register, &read_data_register);
        xil_printf("[I2C_Read n. %d] = %x\n", i, read_data_register);
    }

    sleep(1);

    xil_printf("\n***TESTING IIC READ AT 0x0B FOR %d TIMES***\n", MAX_TEST);
    slave_register = 0x0B;
    for(i = 1; i <= MAX_TEST; i++)
    {
        I2C_read_8_bit(slave_address, slave_register, &read_data_register);
        xil_printf("[I2C_Read n. %d] = %x\n", i, read_data_register);
    }

    xil_printf("\n");
}
```

4.7 Contenuto del file “test.c”

4.1.2 Organizzazione

Per rendere più accessibile e di facile comprensione il progetto riorganizzato, nei capitoli che seguono, vengono mostrate tramite flow-chart le sequenze in cui le varie funzioni vengono invocate.
Questa sezione vuole essere una sorta di documentazione tecnica dettagliata circa l'esatto funzionamento del codice prodotto in fase sperimentale.

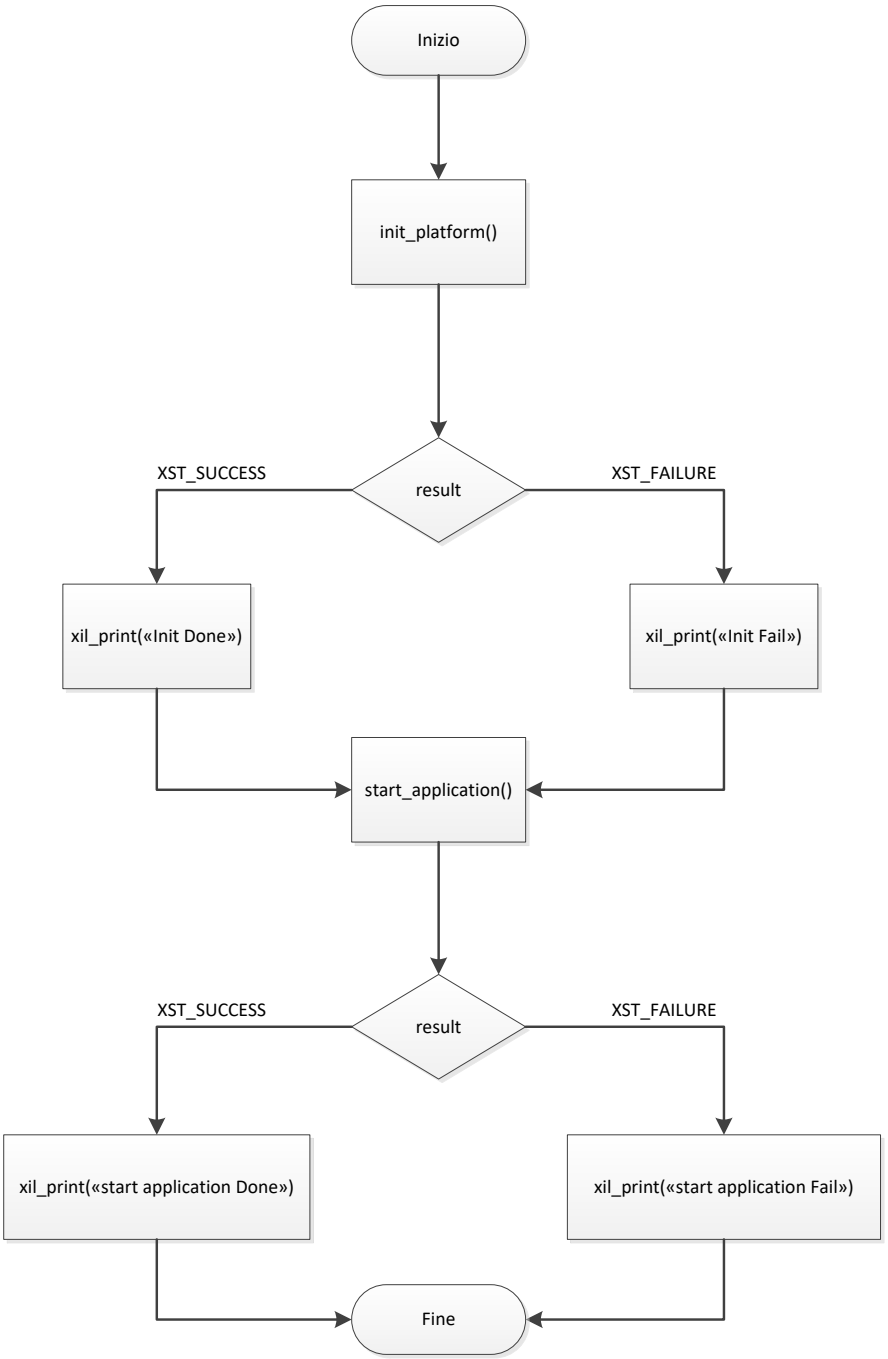
Non essendo possibile organizzare un progetto sviluppato in linguaggio C in classi e package come invece avverrebbe in un qualsiasi linguaggio di programmazione orientato agli oggetti, l'approccio utilizzato durante tutte le fasi di analisi e sviluppo è stato quello del "*divide et impera*", ovvero dividere ogni parte del software in funzioni a granularità molto fine.

Questa scelta permette di riutilizzare buona parte del codice per progetti futuri e, a programmatori con diversa esperienza, di comprendere chiaramente il codice senza bisogno di leggere ogni file o funzione. In caso di modifiche si potrà facilmente eliminare il codice presente all'interno della funzione sostituendolo con il nuovo senza bisogno di apportare pesanti modifiche a tutto il progetto.

Nei linguaggi *Object Oriented* questo problema sarebbe stato risolto mediante astrazione con l'utilizzo di interfacce, ogni soluzione può essere implementata in una classe senza bisogno di commentare o rimuovere il codice originale. Essendo impossibile definire interfacce nei linguaggi di più basso livello come ad esempio il C, l'unica soluzione possibile risulta quindi quella adottata in questo progetto.

4.1.2.1 File "main.c"

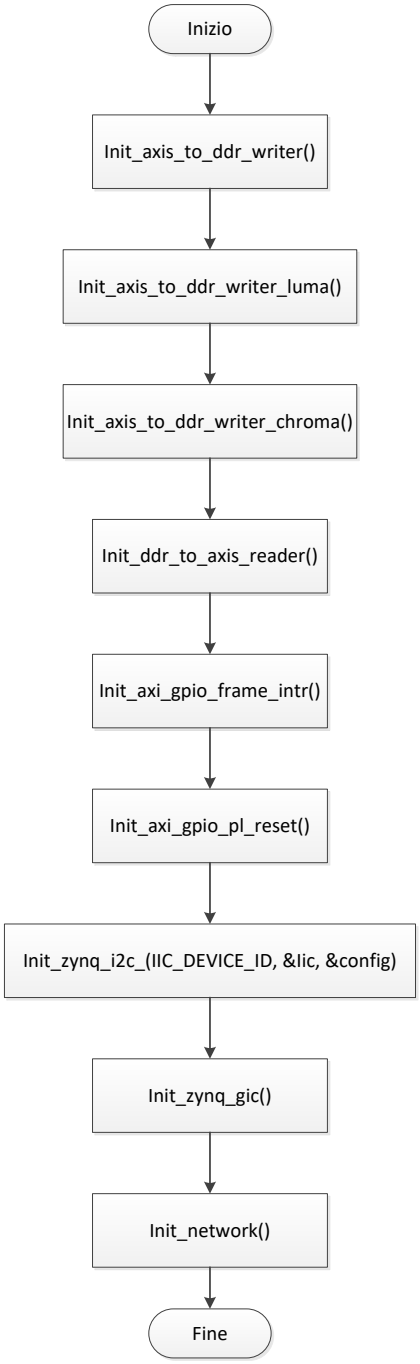
Come anticipato nel capitolo precedente, di seguito è mostrato un diagramma a blocchi che mostra l'esatto ordine di invocazione delle funzioni per rendere il codice ancora più leggibile.



4.8 Sequenza di invocazioni file "main.c"

4.1.2.2 Funzione "init_platform"

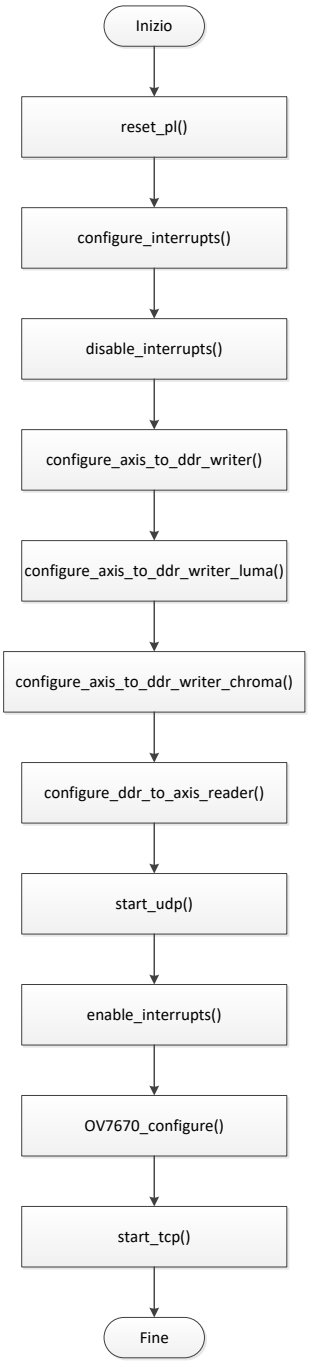
Di seguito è mostrata la sequenza in cui i vari componenti vengono inizializzati come anticipato nel capitolo 4.1.1.4.



4.9 Sequenza di invocazioni funzione "init_platform"

4.1.2.3 Funzione "start_application"

Come anticipato nel capitolo 4.1.1.5, di seguito è mostrata la sequenza in cui i vari componenti vengono configurati e le applicazioni avviate.



4.10 Sequenza di invocazioni funzione "start_application"

4.1.3 Documentazione tramite Doxygen

DoxyGen [3] è un'applicazione per la generazione automatica della documentazione a partire dal codice sorgente di un generico software. È un progetto open source disponibile sotto licenza GPL, scritto per la maggior parte da Dimitri van Heesch a partire dal 1997.

DoxyGen è un sistema multipiattaforma (Windows, Mac OS, Linux, ecc.) e opera con i linguaggi C++, C, Java, Objective C, Python, IDL (versioni CORBA e Microsoft), Fortran, PHP, C#, e D. Il sistema estrae la documentazione dai commenti inseriti nel codice sorgente e dalla dichiarazione delle strutture dati.

Il risultato finale è disponibile sotto forma di pagine HTML oppure nei formati CHM, RTF, PDF, LaTeX, PostScript o man pages di Unix. Il formato HTML prodotto permette al lettore un'agevole navigazione della struttura dei file sorgenti. La documentazione prodotta riporta anche il diagramma delle classi, nei casi in cui sono presenti relazioni di ereditarietà tra strutture dati.

Grazie all'impiego sinergico di DoxyGen con Graphviz, è possibile includere nella documentazione diagrammi delle classi per tutti gli altri tipi di relazioni tra strutture dati. I documenti possono essere generati in diverse lingue. Il sistema è altamente e facilmente configurabile al fine di permettere all'utilizzatore di intervenire su tutti gli aspetti della documentazione prodotta.

È disponibile anche il codice sorgente per permettere particolari modifiche o il *porting* su *host* non previsti tra quelli rilasciati. Una volta scaricati i files, la procedura d'installazione non è complicata, sul sito web ufficiale è disponibile una guida con lo scopo di aiutare gli eventuali programmatori per ogni problema durante la fase di building e/o installazione.

4.1.3.1 Comment Block

I comment blocks rappresentano il perno su cui si regge tutto il processo di traduzione. DoxyGen ricava le informazioni su come costruire il documento dalla semantica del programma sorgente e da commenti speciali che sono stati inseriti nel codice.

Questi commenti speciali sono chiamati "Comment Block": se poi aggiungiamo dei *tag* il Comment Block prende il nome di "Special Documentation Block". Il Comment Block, che utilizza un formato di tipo C o C++, può avere diversi formati:

- È possibile usare uno stile *JavaDoc* [12] inserendo un doppio asterisco in un commento multi-linea:

  ```
  /**
  * … questo è un commento
  */
  ```

- È anche possibile usare una notazione Qt e aggiungere un punto esclamativo dopo l'apertura di un commento secondo lo stile C, come:

  ```
  /*!
  * … questo è un commento
  */
  ```

- Nei due casi l'asterisco in posizione intermedia è opzionale:

  ```
  /*!
  … questo è un commento
  */
  ```

- Una terza alternativa è di utilizzare la seguente notazione:

  ```
  ///
  /// … questo è un commento
  ///
  ```

- Oppure:

  //!
  //! … questo è un commento
  //!

- Altro modo (spesso utilizzato in programmazione web) è il seguente:

  ////////////////////////////////
  /// … questo è un commento
  ////////////////////////////////

4.1.3.2 Tag

Il tag è un comando speciale ed è inserito nel codice sorgente anteponendoci un carattere identificativo. I tag inseriti nel codice iniziano con un backslah "\" o con il simbolo "@".

Alcuni comandi possono poi avere uno o più argomenti e l'intervallo di un argomento può essere così rappresentato:

- Se l'argomento è rappresentato come <argomento> allora il campo argomento è una singola parola.
- Se, invece, è utilizzata la notazione (argomento), allora il campo argomento viene esteso fino alla fine della linea sul quale si trova il commento.
- Usando la notazione {argomento} allora il campo argomento viene esteso fino al prossimo paragrafo. I paragrafi sono delimitati da una linea bianca o da un indicatore di sezione.
- Usando la notazione [argomento] indichiamo che l'argomento stesso è opzionale.

Tra i tag principali utili per la generazione della manualistica troviamo:

- *@brief description*, questo tag fornisce una breve descrizione che sarà inserita nel file d'uscita, spesso si inserisce una spiegazione generale sulla funzione.
- *@author name1*, con questo tag nel file d'uscita ci sarà la stampa del nome dell'autore.
- *@version VersionNumber*, questo tag stamperà nel file d'uscita il numero di versione.
- *@file filename*, questo tag stamperà nel file d'uscita il nome del file come header della pagina.

Altri tag meno utilizzati, ma di particolare interesse sono:

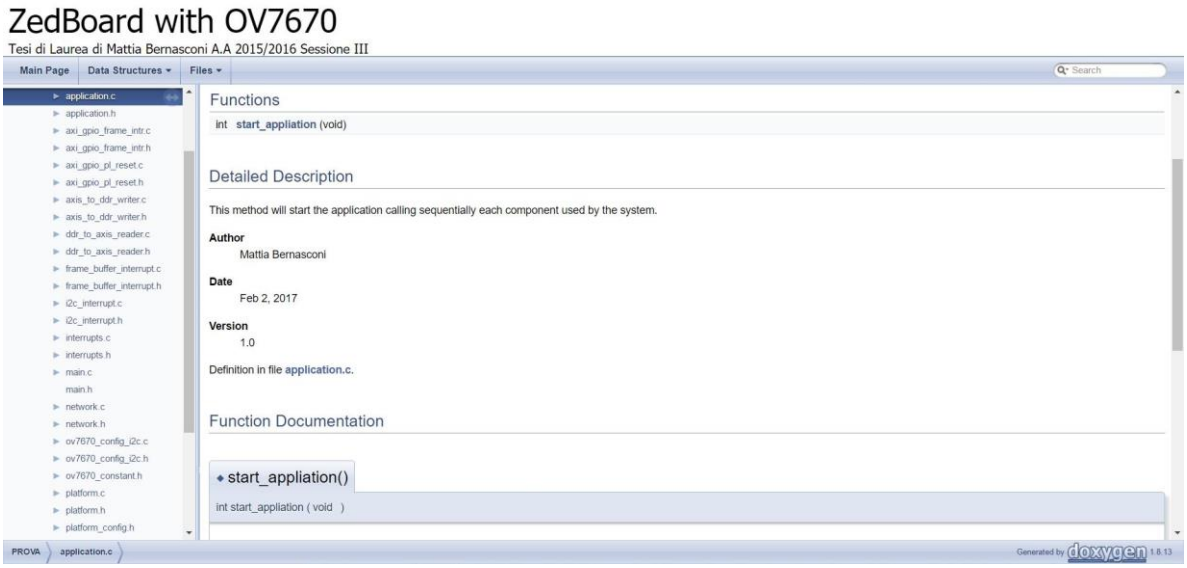
- *@warning*, per eventuali messaggi di warning.
- *@bug*, per una eventuale descrizione di bugs.
- *@see FunctionA()*, per indicare all'utente di fare riferimento ad altra funzione.

4.1.3.3 Output

Per procedere alla generazione della documentazione è necessario usare il comando con il file di configurazione associato al progetto (il *doxyfile* deve essere stato creato e configurato in precedenza tramite interfaccia grafica o linea di comando). A questo punto con DoxyGen si ottiene la documentazione con tutti i suoi file associati nella cartella come deciso nel file di configurazione.

```c
/**
 * @file application.c
 * @brief This method will start the application calling sequentially each component used by the system.
 *
 * @author Mattia Bernasconi
 *
 * @date Feb 2, 2017
 * @version 1.0
 *
 */

#include "application.h"

/**
 * @author Mattia Bernasconi
 * @param void No parameters needed
 * @date 02/02/2017
 */
int start_appliation(void)
{
    int result;

    ///Functions call order:
    ///1)RESET PL
    result = reset_PL();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about reset_PL\n\r");
    }
    xil_printf("reset_PL done \n\r");

    ///2)CONFIGURE INTERRUPTS
    result = configure_interrupts();
    if(result != XST_SUCCESS)
    {
        xil_printf("There is an error about configure_interrupts\n\r");
    }
    xil_printf("configure_interrupts done \n\r");
```

4.11 Esempio di file sorgente con tag specifici



4.12 Esempio di output della documentazione generata

4.2 Lato client multi OS

Lato client, il software per la ricezione delle immagini, è stato riprogettato e riscritto completamente. Avendo introdotto lato board la trasmissione di tutte le immagini acquisite, nello specifico i frame post elaborazione tramite filtri, i frame circa la componente di luminanza ed i frame contenenti le componenti di crominanza, è stato necessario cambiare buona parte del client per poter correttamente ricevere e visualizzare tutti i flussi video.

Nella versione esistente del client era prevista la ricezione di un pacchetto dati tramite protocollo *UDP* contenente una struttura dati al cui interno era presente un array di byte riempito con i dati delle immagini trasmesse.

Avendo però introdotto anche la trasmissione di altri due flussi video per poter mostrare le immagini a colori acquisite dai sensori *OV7670* è stato necessario introdurre anche nel software del cliente i due nuovi vettori in maniera tale da allocare correttamente la memoria minima indispensabile ove salvare i dati ricevuti dal broadcast *UDP*.

Avendo poi introdotto lato board anche la ricezione di comandi tramite protocollo *TCP* è risultato indispensabile progettare un client in grado di inviare segnali tramite questo protocollo per poter verificare il corretto funzionamento di questa nuova caratteristica.

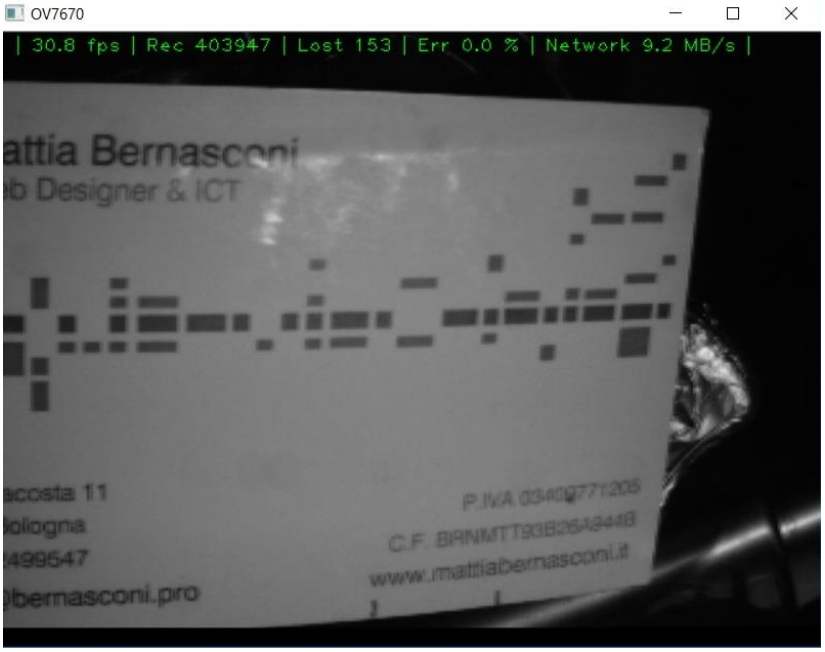
I flussi video acquisiti tramite il protocollo *UDP* sono salvati in memoria tramite una “memcpy” e successivamente convertiti in immagini tramite l’utilizzo delle librerie open source OpenCV. I dati contenenti le immagini salvate in memoria sono convertiti in “IplImage” in maniera tale che la funzione di libreria “cvShowImage” sia in grado di mostrare i frame sotto forma di video in tempo reale.

Prima di passare il flusso video alla funzione “cvShowImage”, questo è elaborato e convertito in uno spazio di colore RGB per poter inserire alcuni dati in sovraimpressione alle immagini mostrate. I dati di interesse da mostrare vengono posizionati nella parte alta di tutte le immagini e aggiornati ad ogni nuovo frame mostrato.

I dati che si è scelto di visualizzare sono:

- Frame rate nell’unità di misura “fps” ovvero numero di frame ricevuti in un secondo
- Numero totale di frame ricevuti
- Numero totale di frame persi o arrivati a destinazione corrotti

- Percentuale di errore sui frame persi
- Velocità di trasmissione delle immagini tramite ethernet in MB/s



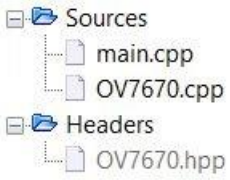
4.13 Esempio di overlay dei dati sulle immagini mostrate a video

Altra funzionalità introdotta nell'implementare il nuovo client di ricezione è la possibilità di salvare tutti i frame (quello elaborato dalla FPGA, la componente "luma" e "chroma" delle immagini originali) tramite la pressione del tasto "S" della tastiera. Se durante un qualsiasi momento in cui la visualizzazione delle immagini è attiva l'utente preme il tasto in questione, il software genera tre immagini in formato *PNG* e le salva all'interno del direttorio in cui risiede l'eseguibile (file binario) del client. Tengo a precisare che i file *PNG* che sono salvati contengono l'immagine originale acquisita e non quella con i dati i sovraimpressione.

4.2.1 Struttura

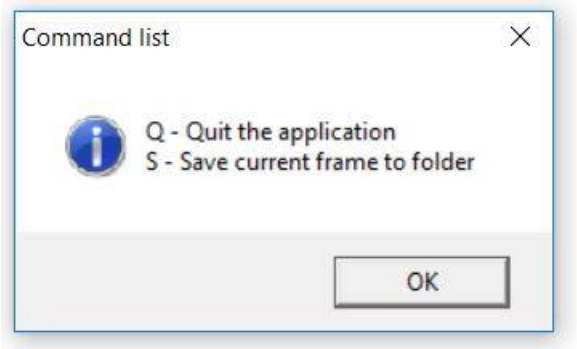
Il client che si occupa di ricevere i pacchetti trasmessi in *broadcast* tramite protocollo UDP è stato ottimizzato e compattato il più possibile in maniera tale da non appesantire il sistema operativo e rendere il codice dello stesso lineare e di facile comprensione.

Il ”*main*” si occupa semplicemente di invocare la funzione “*start_OV7670()*” che metterà la scheda di rete ethernet in ascolto per la ricezione dei dati trasmessi dalla board. Una volta effettuato correttamente il “*bind*” il software è dunque pronto a ricevere i frame e salvarli nella memoria precedentemente allocata.



4.14 Struttura del progetto client

Come si può notare in figura 4.14 i files del progetto sono con estensione “*cpp*” e “*hpp*”, i quali sono caratteristici del linguaggio C++. Essendo il linguaggio C e C++ molto vicini come semantica e sintassi, tutto il codice scritto all’interno dei due files è in realtà codice C. L’area di sviluppo è principalmente sotto piattaforma Windows [13], è stata fatta questa scelta in maniera tale da poter utilizzare le funzioni di libreria “*windows.h*” che vengono utilizzate semplicemente per mostrare a video dei suggerimenti all’utente come mostrato in figura 4.15.



4.15 Messaggio contenente l’elenco dei comandi disponibili

4.2.2 Organizzazione

Come anticipato nei capitoli precedenti il client è stato semplificato il più possibile per non appesantire il sistema operativo e per permettere la portabilità tra diversi sistemi operativi.

L'ambiente di sviluppo scelto ed utilizzato per la scrittura ed il debug del client è CodeBlocks.

Essendo quella discussa in questa tesi una versione ottimizzata per Windows, si è scelto di non utilizzare le librerie native di questo sistema operativo per la gestione del network, ma di utilizzare un pacchetto di librerie più leggero. Nello specifico sono state utilizzate le funzioni offerte dal pacchetto di librerie *WinSock* [14].

Come anticipato e mostrato in figura 4.16 il "*main*" risulta molto snello e con pochissime istruzioni. È eseguita un'unica operazione che è invocare l'avvio vero e proprio dell'applicativo tramite la funzione "*start_OV7670()*" la cui definizione è contenuta nel file "*OV7670.h*".

Tutto ciò per poter permetter in futuro di aggiungere prima o dopo l'avvio dell'applicazione, altre istruzioni senza dover stravolgere il codice esistente. Se tutta la logica del programma fosse stata scritta all'interno del "*main*", come nel progetto originale, ogni piccola modifica avrebbe richiesto ai nuovi programmatori di capire a fondo la sequenza delle invocazioni già esistenti e avrebbe potuto creare confusione in caso di nomi di variabili già esistenti o poco "autoesplicative".

```
#include <iostream>
#include <windows.h>
#include "OV7670.hpp"

using namespace std;

int main(int argc, char **argv)
{
    // Start the network udp client
    // and the OpenCV Visualizer
    start_OV7670();

    return 0;
}
```

4.16 Contenuto del file "main.c" del software client

4.2.2.1 File "OV7670.h"

Per poter correttamente utilizzare le funzioni delle librerie OpenCV e WinSock è stato necessario importarle all'interno del progetto. Tale operazione è effettuata tramite direttive al preprocessore grazie alla keyword "include". Tutte le istruzioni di inclusione per il preprocessore sono state inserite all'interno del file di intestazione "*OV7670.h*" come mostrato in figura 4.17.

Come già evidenziato nel capitolo 4.1 si può notare che la struttura "*packet_data*" è identica a quella definita nel progetto lato board.

```
#include <stdio.h>
#include <WinSock2.h>
#include <direct.h>
#include <sys/time.h>

#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <stdlib.h>

#include <opencv2/core.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc.hpp>
//#include <opencv2/imgcodecs.hpp>
#include "opencv2/highgui.hpp"

//#define WIN32_LEAN_AND_MEAN
#define WIDTH 640
#define HEIGHT 480
#define FRAME_SIZE 307200
#define FRAME_I_SIZE 13520  //30720 //7680

typedef unsigned char BYTE;

typedef struct {
    int count;
    int fragment;
    int frame_index;
    BYTE data[FRAME_I_SIZE];
    BYTE data_luma[FRAME_I_SIZE];
    BYTE data_chroma[FRAME_I_SIZE];
} packet_data;

int start_OV7670();
```

4.17 Contenuto del file "OV7670.h" del software client

4.2.2.2 File "OV7670.cpp"

All'interno di questo file è stata implementata sia la logica riguardante la ricezione dei dati tramite protocollo *UDP*, sia la conversione delle immagini ricevute nello spazio *RGB* per poter inserire in sovraimpressione (*overlay*) i dati mostrati nell'immagine. 4.13

```
#include "OV7670.hpp"
#pragma comment(lib, "ws2_32.lib")

using namespace cv;

IplImage* image;
IplImage* image_luma;
IplImage* image_chroma;

IplImage* image_text;
IplImage* image_luma_text;
IplImage* image_chroma_text;

CvFont font;
CvScalar color;
```

4.18 Inizio del file "OV7670.cpp"

La figura 4.18 mostra che sono state definite sei variabili "*IplImage*", all'interno delle prime tre saranno inserite le immagini originali ricevute dalla board, mentre all'interno delle altre tre saranno presenti le immagini con i dati in sovraimpressione.

Le variabili "*CvFont*" e "*CvScalar*" sono state invece utilizzate per decidere il font e il colore delle scritte poste in *overlay* sulle immagini.

Sono state inoltre definite due funzioni di appoggio per poter gestire più facilmente l'inizializzazione delle variabili "*IplImage*" e per poter inserire una qualsiasi stringa sui frame mostrati a video. Come mostrato nell'immagine 4.19 le funzione che si occupa della configurazione delle variabili è la "*init_my_opencv()"*, mentre la funzione che si occupa di inserire il testo sulle immagini è la "*write_text(...)*".

```
void write_text(char * msg, IplImage* image_source, IplImage* image_dest)
{
    cvCvtColor(image_source, image_dest, CV_GRAY2BGR);
    cvPutText(image_dest, msg, cvPoint(10,15), &font, color);
}

void init_my_opencv()
{
    cvInitFont(&font, CV_FONT_HERSHEY_PLAIN, 1, 1, 0, 1 , 8);
    color = cvScalar(0,255,0,0);
    image = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 1);
    image_luma = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 1);
    image_chroma = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 1);

    image_text = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 3);
    image_luma_text = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 3);
    image_chroma_text = cvCreateImage(cvSize(WIDTH, HEIGHT), IPL_DEPTH_8U, 3);
}
```

4.19 Funzioni "write_text(...)" e "init_my_opencv()"

Per mostrare i dati in sovraimpressione alle immagini, è stato necessario calcolarti tramite le prime istruzioni mostrate in figura 4.20. Dopo aver calcolato ed inserito i dati all'interno della stringa "*msg*", vengono copiate le immagini all'interno della memoria tramite una "*memcpy*", è invocata la funzione "*write_text*" per inserire la stringa sui frame ed infine tramite la funzione "*cvShowImage(...)*" viene mostrato il flusso video.

```
/*Start create message to show over image*/
frame_ricevuti++;
gettimeofday(&end, NULL);
frame_rate = 1000000.0/(end.tv_usec - start.tv_usec);
gettimeofday(&start, NULL);
p_frame_persi = (100.0*frame_persi)/(frame_ricevuti);
byte_rate = (FRAME_SIZE*frame_rate)/1024000.0;
sprintf(msg, "| %.1f fps | Rec %d | Lost %d | Err %.2f %% | Network %.1f MB/s |", frame_rate, frame_ricevuti, frame_persi, p_frame_persi, byte_rate);
/*End create message*/

//Show Image NORMAL
memcpy(image->imageData, frame, sizeof(BYTE)*WIDTH*HEIGHT);
write_text(msg, image, image_text);
cvShowImage("OV7670", image_text);

//Show Image LUMA
memcpy(image_luma->imageData, frame_luma, sizeof(BYTE)*WIDTH*HEIGHT);
write_text(msg, image_luma, image_luma_text);
cvShowImage("OV7670 Luma", image_luma_text);

//Show Image CHROMA
memcpy(image_chroma->imageData, frame_chroma, sizeof(BYTE)*WIDTH*HEIGHT);
write_text(msg, image_chroma, image_chroma_text);
cvShowImage("OV7670 Chroma", image_chroma_text);
```

4.20 Calcolo dei dati da mostrate in sovraimpressione e visualizzazione a video delle immagini

Per aiutare l'utente a terminare l'applicazione, e nell'eventuale salvataggio dei frame, è stata utilizzata la funzione "*cvWaitKey(...)*" che attende la pressione di un tasto e lo associa alla variabile "*key*". Una volta rilevata la pressione di un tasto con una semplice struttura condizionale è possibile gestire le diverse condizioni.

Nel mio lavoro di tesi è stata implementata la gestione di solo due tasti:

- "S" Sono salvati i tre frame correnti sotto forma di immagini *PNG* della dimensione *640x480* all'interno del direttorio in cui è presente il file eseguibile binario.
- "Q" È chiusa la connessione di rete, vengono rilasciate le variabili che contengono le immagini e viene liberata la memoria allocata dinamicamente tramite la primitiva "*free(...)*"

```
//KEY PRESSED
char key = cvWaitKey(1);

if(key == 's')//Save Frame
{
    cvSaveImage("NORMAL.png", image);
    cvSaveImage("LUMA.png", image_luma);
    cvSaveImage("CHROMA.png", image_chroma);

    //Confirmation Dialog Box
    MessageBox(NULL, "Images Correctly Saved!", "Success!", MB_ICONINFORMATION | MB_OK | MB_DEFBUTTON2);
}
else if (key == 'q')//Quit the application
{
    printf("Chiusura ricezione...\n");

    //Release all memory
    cvReleaseImage(&image);
    cvReleaseImage(&image_text);
    cvReleaseImage(&image_luma);
    cvReleaseImage(&image_luma_text);
    cvReleaseImage(&image_chroma);
    cvReleaseImage(&image_chroma_text);

    free(frame);
    free(frame_chroma);
    free(frame_luma);

    free(tmp);
    free(msg);

    printf("Terminato...\n");

    closesocket(sd);
    WSACleanup();

    exit(0);
}
```

4.21 Gestore eventi in base al tasto premuto dall'utente

Capitolo 5 - Implementazione di comunicazioni mediante il protocollo TCP

5.1 Protocollo TCP per comunicazioni sicure

Il protocollo TCP può essere classificato al livello trasporto del modello di riferimento *OSI*, ed è solitamente usato in combinazione con il protocollo di livello rete IP. La corrispondenza con il modello OSI non è perfetta, poiché il TCP e l'IP nascono prima di questo modello. La loro combinazione è indicata come TCP/IP ed è a volte erroneamente considerata un unico protocollo.

In linea con le regole del livello di trasporto stabiliti dal modello *ISO-OSI*, e con l'intento di superare il problema della mancanza di affidabilità e controllo della comunicazione sorto con l'interconnessione su vasta scala di reti locali in un'unica grande rete geografica, TCP è stato progettato e realizzato per utilizzare i servizi offerti dai protocolli di rete di livello inferiore (IP e protocolli di livello fisico e livello datalink) che definiscono efficacemente il modo di trasferimento sul canale di comunicazione, ma che non offrono alcuna garanzia di affidabilità sulla consegna in termini di ritardo, perdita ed errore dei pacchetti informativi trasmessi, sul controllo di flusso tra terminali e sul controllo della congestione di rete, supplendo quindi ai problemi di cui sopra e costruendo così un affidabile canale di comunicazione tra due processi applicativi di rete.

Il canale di comunicazione così costruito è costituito da un flusso bidirezionale di byte a seguito dell'instaurazione di una connessione agli estremi tra i due terminali in comunicazione. Inoltre alcune funzionalità del TCP sono vitali per il buon funzionamento complessivo di una rete IP. Sotto questo punto di vista il TCP può essere considerato come un elemento di rete che si occupa di garantire una qualità di servizio minima su una rete IP che a livello sottostante è di tipo *best-effort*.

Le principali differenze tra TCP e UDP (User Datagram Protocol) sono:

- TCP è un protocollo orientato alla connessione. Pertanto, per stabilire, mantenere e chiudere una connessione, è necessario inviare pacchetti di servizio i quali aumentano *l'overhead* di comunicazione. Al contrario, UDP è senza connessione e invia solo i datagrammi richiesti dal livello applicativo.
- UDP non offre nessuna garanzia sull'affidabilità della comunicazione ovvero sull'effettivo arrivo dei segmenti, sul loro ordine in sequenza in arrivo; al contrario il TCP tramite i meccanismi di *acknowledgement* e di ritrasmissione su *timeout* riesce a garantire la consegna dei dati, anche se al costo di un maggiore overhead.
- L'oggetto della comunicazione di TCP è il flusso di byte mentre quello di UDP è il singolo datagramma.

5.2 Implementazione del protocollo

Per migliorare l'implementazione del protocollo TCP/IP sulla board, e non appesantire troppo il sistema, sono state utilizzate le funzioni messe a disposizione dalle librerie LightWeight *TCP/IP*, meglio note come *lwIP* [18], sviluppate da Adam Dunkels presso "Swedish Institute of Computer Science".

Queste librerie nascono come un'implementazione ridotta, ma con funzionalità più che sufficienti, per lo sviluppo del progetto trattato in questa tesi. Gli obiettivi di questa implementazione sono l'ottimizzazione delle risorse utilizzate, in maniera tale da aumentare l'efficienza e ridurre i consumi della piattaforma hardware che ha il compito di mandare in esecuzione il codice compilato dal sorgente.

Essendo le caratteristiche di queste librerie ottimali per progetti embedded, si sono rivelate molto utili nel nostro caso; avendo a disposizione una quantità di banda elevata, se l'invio dei dati non avvenisse abbastanza velocemente potrebbero verificarsi problemi nella gestione degli *interrupt*.

Per poter essere utilizzata, la libreria lwIP deve essere inizializzata. A tal proposito è fondamentale richiamare il metodo "*lwip_init(...)*" prima di utilizzare la libreria. Essendo queste librerie un'implementazione standard del protocollo TCP/IP, che mette a disposizione molti parametri di configurazione per ottimizzare le risorse, è importante eseguire una corretta gestione della memoria, in particolare per quanto riguarda l'utilizzo dei "*pbuf*". Compito del programmatore è la completa gestione dell'allocazione e del rilascio delle risorse utilizzate.

È inoltre possibile modificare le dimensioni dei buffer di ricezione e invio, oltre che ai *timeout* e la priorità di gestione di diversi tipi di pacchetti (distinti per protocollo o "*netif*"). Queste caratteristiche rendono lwIP molto versatile e in grado di adattarsi a utilizzi che richiedono specifiche differenti.

5.2.1 Funzione "start_tcp"

Come anticipato nel capitolo precedente per poter utilizzare le librerie lwIP bisogna includerle all'interno del progetto tramite istruzioni per il preprocessore ed inizializzare tutte le variabili richieste.

La gestione delle richieste TCP è stata implementata in due files:

- "*tcp_connection.c*" che si occupa di attendere la connessione da parte di un cliente remoto, accettarla, gestirla ed infine chiuderla.
- "tcp_connection.h" in cui sono definite le variabili utilizzate quali ad esempio la porta su cui ascoltare le eventuali connessioni e le direttive per il preprocessore.

```
// Intialise the TCP connection to be used
int start_tcp(void)
{
    xil_printf("[TCP] Starting TCP server on port %d\n", TCP_PORT);

    // Create a new TCP Protocol Cotrol Buffer abbreviated as PCB.
    pcb_tcp = tcp_new();

    // Bind the PCB to a Port and any IP address.
    tcp_bind(pcb_tcp, IP_ADDR_ANY, TCP_PORT);

    // Put the PCB and TCP connection on Board in listening state
    pcb_tcp = tcp_listen(pcb_tcp);

    xil_printf("[TCP] Listening on port %d\n\n", TCP_PORT);

    // This function specifies the callback function that will be called when a client asks for connection with board
    tcp_accept(pcb_tcp, connection_accept);

    return XST_SUCCESS;
}
```

5.1 Implementazione della funzione "start_tcp"

La funzione mostrata in figura 5.1 si occupa di creare il *PCB* (Protocol Control Buffer), eseguire il bind del *PCB* creato alla porta indicata nel file di intestazione, mettere in ascolto il canale verso un qualsiasi host remoto e definire la funzione di *callback* che sarà invocata non appena una connessione TCP viene stabilita.

5.2.2 Funzione "connection_accepted"

Non appena un host remoto si collega alla board per inviargli segnali o comandi, viene accettata la connessione tramite la funzione mostrata in figura 5.2.

Sono inoltre settate alcune proprietà tramite l'istruzione "*LWIP_UNUSED_ARG(...)*" e definite le priorità tramite la funzione "*tcp_setprio(...)*" .

È successivamente chiamata la funzione spiegata nel capitolo 5.2.3 che avrà il compito di ricevere i dati e le istruzioni contenute nel pacchetto *TCP*.

```
// This function gets called for accepting incoming TCP connections.
static err_t connection_accept(void *arg, struct tcp_pcb *pcb_tcp, err_t err)
{
    print_ip("\n[TCP] Connection accepted from ", &pcb_tcp->remote_ip);

    LWIP_UNUSED_ARG(arg);
    LWIP_UNUSED_ARG(err);

    tcp_setprio(pcb_tcp, TCP_PRIO_MIN);

    // Specifies the function to call, on reception of data on TCP
    tcp_recv(pcb_tcp, tcp_data_receive);

    tcp_err(pcb_tcp, NULL);
    tcp_poll(pcb_tcp, NULL, 4);

    return ERR_OK;
}
```

5.2 Implementazione della funzione "connection_accepted"

5.2.3 Funzione "tcp_data_receive"

Questa funzione è quella che si occupa di ricevere il *payload* (i.e., i dati) contenuti all'interno dei pacchetti *TCP*, ed eventualmente gestire le richieste ricevute. Se il buffer non è vuoto, e se non si sono verificati errori, si procede con la lettura dei dati dal canale. Una volta letti i dati, in questa implementazione è trasmessa una stringa di testo, si potrebbero prevedere diversi comportamenti in base alle istruzioni ricevute.

Per eseguire operazioni e istruzioni diverse in base al tipo di dato ricevuto basterà quindi aggiungere un semplice blocco condizionale, prevedendo tutti i possibili comandi inviati dal client remoto, e richiamare semplicemente la funzione associata al comando. Nel codice riportato in figura 5.3 non vi è alcuna gestione delle istruzioni, ma i dati ricevuti vengono stampati a video tramite monitor seriale.

```
// This function gets called on reception of data on TCP.
static err_t tcp_data_receive(void *arg, struct tcp_pcb *pcb_tcp, struct pbuf *pbuffer, err_t err)
{
    unsigned int len;
    unsigned char *pc;

    // Check if no error has occured and packet buffer is not empty
    if (err == ERR_OK && pbuffer != NULL)
    {
        //When the application has taken the data, it has to call the tcp_recved()
        //function to indicate that TCP can advertise increase the receive window.(N.B. As specified by LWIP).

        tcp_recved(pcb_tcp, pbuffer->tot_len);

        // Get the pointer to the Payload area of received TCP packet
        pc = (unsigned char *) pbuffer->payload;

        // Get the length of the data payload and assign it to our data count variable
        len = pbuffer->tot_len;

        // The data can be taken in this part as "pc" will contain the starting address of the payload. Just use the "pc" pointer and get the data in your buffer.
        xil_printf("[TCP] %d Bytes of data receive: '%s'\n", len, pc);

        // Free the packet buffer
        pbuf_free(pbuffer);

    }
    else
    {
        // Free the packet buffer
        pbuf_free(pbuffer);
    }

    // Close connection if empty packet and no error
    if (err == ERR_OK && pbuffer == NULL)
    {
        close_conn(pcb_tcp);
    }

    return ERR_OK;
}
```

5.3 Implementazione della funzione "tcp_data_receive"

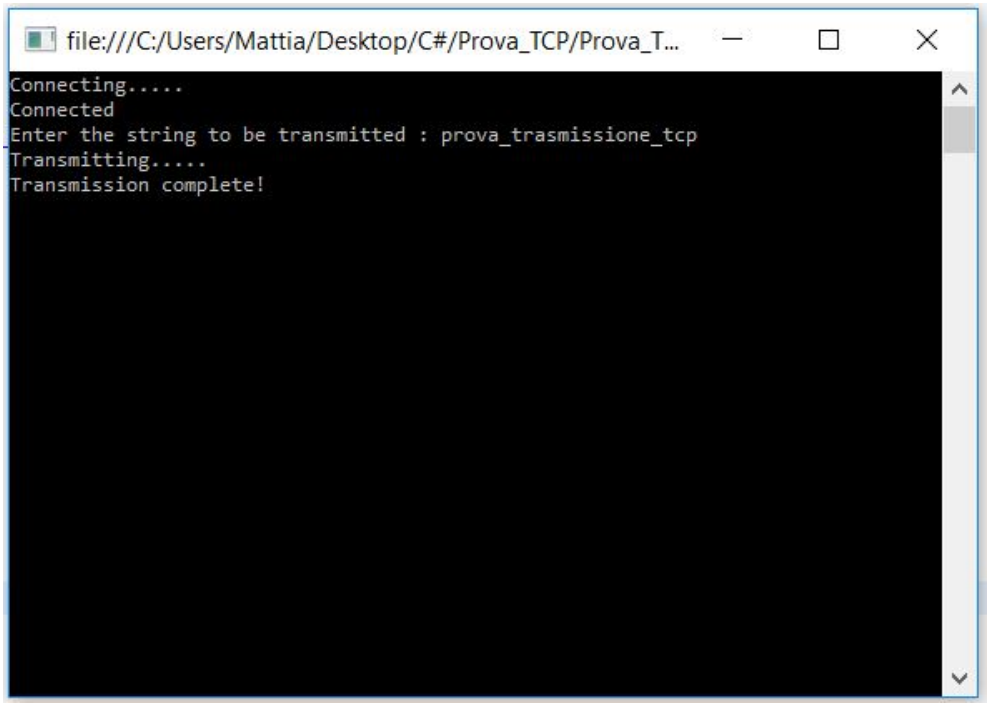
5.2.4 Funzione "close_conn" e risultati

Una volta ricevuti i dati dal client remoto la connessione viene chiusa, e le risorse utilizzate vengono liberate. A questo punto sia la connessione sia la comunicazione sono stabilite per mezzo di thread, un nuovo cliente può adesso connettersi alla board ed il processo di ricezione dati ricomincia.

```
// Function for closing the connection.
static void close_conn(struct tcp_pcb *pcb_tcp)
{
    tcp_arg(pcb_tcp, NULL);
    tcp_sent(pcb_tcp, NULL);
    tcp_recv(pcb_tcp, NULL);
    tcp_close(pcb_tcp);
    print_ip("[TCP] Connection closed with client ", &pcb_tcp->remote_ip);
}
```

5.4 Implementazione della funzione "close_conn"

In figura 5.5 è mostrata la trasmissione di una semplice stringa di testo alla board tramite client da console.



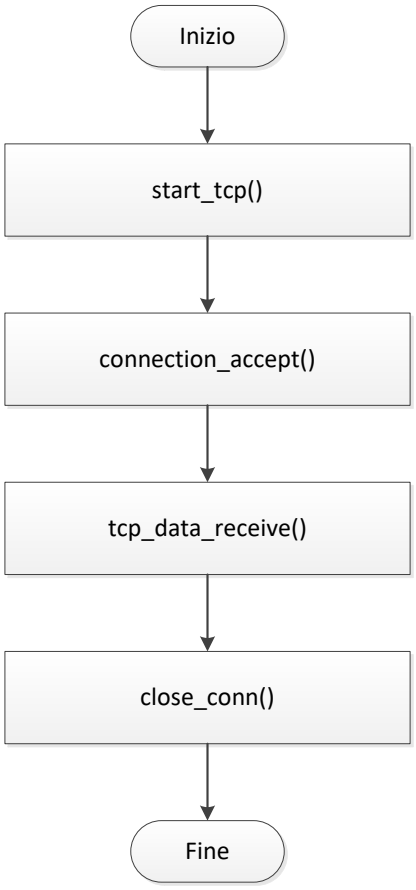
5.5 Esempio di trasmissione stringa tramite TCP

In figura 5.6 è invece mostrato il contenuto del pacchetto ricevuto tramite protocollo TCP, come detto in precedenza in questa prima implementazione i dati sono semplicemente stampati a video su monitor seriale senza subire alcuna variazione.

5.6 Esempio di ricezione stringa tramite TCP

5.2.5 Ordine delle invocazioni

Per una maggiore chiarezza e leggibilità del codice, di seguito è mostrato un flow-chart circa l'esatta sequenza in cui vengono progressivamente chiamate le funzioni per la corretta gestione e ricezione di dati tramite protocollo *TCP*.



4.8 Sequenza di invocazioni file "tcp_connection.c"

Capitolo 6 - Sperimentazione

6.1 Risultati sperimentali

Dopo aver riorganizzato l'esistente progetto e aver risolto alcuni problemi sulla sincronizzazione dell'invio dei frame acquisiti tramite i sensori OV7670 sono state introdotte tre principali nuove funzionalità: gestione degli interrupts annidati, trasmissione delle immagini sia in scala di grigi dopo essere state elaborate dalla board, sia nelle componenti di luminanza e crominanza così come acquisite dalla telecamera.

L'aver introdotto la trasmissione delle due nuove componenti necessarie per ricostruire le immagini a colori ha ovviamente richiesto che quest'ultime siano inviate dalla ZedBoard al cliente remoto. Il che ha determinato un decisivo incremento dell'utilizzo della banda a disposizione poiché il traffico dati trasmesso risulta essere quasi triplicato. Questo aumento ha purtroppo causato anche un lieve innalzamento della percentuale di frame (o meglio datagrammi) persi o corrotti che risulta essere di 1 ogni 1800 (dunque uno al minuto) circa. Si faccia riferimento alla figura 6.1 per i dati acquisiti.

Per la natura stessa della progettazione e implementazione del protocollo UDP, le informazioni trasmesse con questo sistema vengono incapsulate in datagrammi (spesso frammentati in più pacchetti). Essendo UDP un sistema di comunicazione best-effort non è quindi garantito né che i pacchetti trasmessi siano ricevuti nello stesso ordine né che questi siano effettivamente consegnati a destinazione.

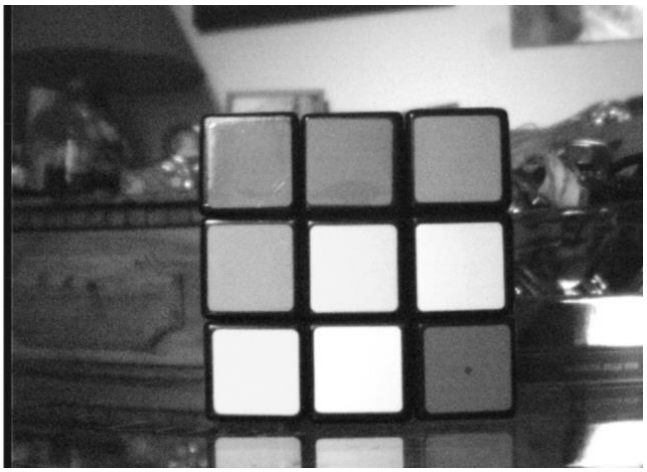
Avendo quasi triplicato il numero di byte, il numero di frammenti dei datagrammi inviati risulta anch'esso aumentato notevolmente. Per mantenere una compatibilità tra i vari sistemi operativi è stato necessario ridurre le dimensioni dei frammenti a 13.520 (contro i 30.720 del progetto precedente).

È inoltre impossibile comprimere le immagini per ridurre lo spazio occupato e riacquisire un frame già trasmesso (la immagini sono trasmesse in real-time e soprattutto una compressione *lossy* pregiudicherebbe successive elaborazioni da parte di algoritmi di computer vision) risulta quindi difficile abbassare il numero di pacchetti persi durante la trasmissione. Per abbattere l'attuale percentuale di errore si potrebbe scrivere un driver ad-hoc per la scheda di rete della board e del client. Questa operazione risulta però dispendiosa in termini di tempo e risorse, la soluzione migliore potrebbe essere sfruttare il protocollo TCP per controllare la board e richiedere la trasmissione delle immagini a colori solo quando deciso dall'utente oppure fare un *tuning* sui parametri della trasmissione al fine di ridurre il numero di frame persi.

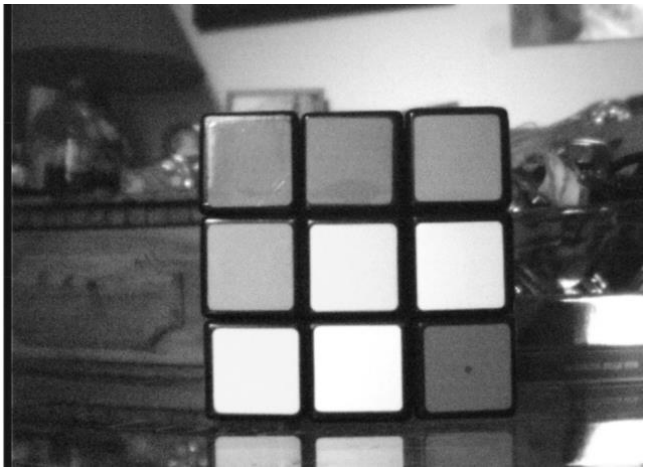
Per ottenere i risultati descritti in precedenza si è rilevato indispensabile il lavoro svolto nel capitolo 4.2, poiché la raccolta dati è stata possibile soltanto grazie alle sovraimpressione degli stessi sulle immagini ricevute dal cliente.

Altro traguardo importante raggiunto in questo lavoro di tesi è stata l'introduzione del protocollo TCP, grazie al quale sarà possibile controllare il funzionamento della board da remoto e senza bisogno di agire fisicamente sull'hardware. L'implementazione di questo metodo di connessione è risultata abbastanza semplice anche se per giungere all'obbiettivo è stato prima necessario studiare a fondo sia l'esistente progetto, sia le librerie lwIP. Questo protocollo è uno tra i più affidabili e robusti attualmente disponibili (specialmente per progetti embedded), questa funzionalità permetterà di inserire in futuro controlli remoti e di ricevere con tutta sicurezza l'esito della richiesta effettuata ed eseguita dalla board.

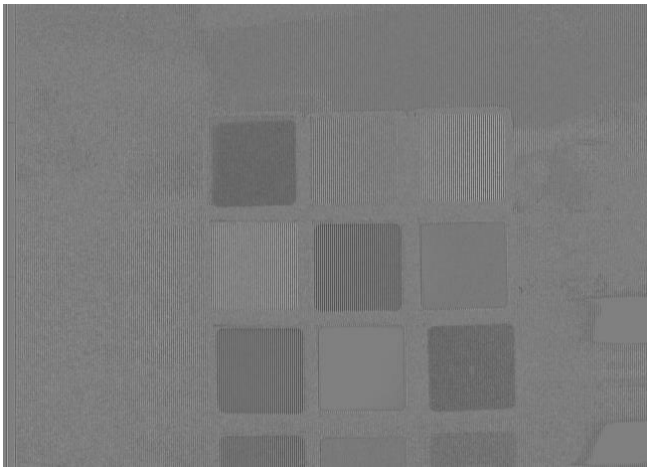
6.1 Dati registrati dal client UDP dopo 225 minuti di funzionamento



6.2 Immagine grayscale con elaborazione su FPGA



6.3 Immagine contenente la componente di luminanza



6.4 Immagine contenete la componente di crominanza

6.5 Immagine nello spazio di colore RGB

6.2 Possibili sviluppi futuri

Oltre alle potenzialità offerte dalla board, e grazie al continuo sviluppo di questo sistema, sarà in futuro possibile aggiungere nuovi componenti per completare ed integrare le funzionalità attualmente presenti.

Uno degli aspetti progettuali trattati in fase di discussione e organizzazione era l'inserimento di nuovi sensori, come accelerometri e giroscopi, per poter visualizzare sulle immagini i dati acquisiti da questi dispositivi e dare all'utente un'idea più precisa circa la reale posizione della telecamera rispetto al suolo o al sistema di riferimento adottato.

Altro aspetto interessante di possibile realizzazione futura potrebbe essere l'arricchimento delle immagini tramite elementi non percepibili mediante i normali sensi umani. Sfruttare questo sistema per cogliere sfumature e dettagli non percepibili normalmente è oggi uno degli argomenti di grande interesse sia in ambito accademico sia in ambito lavorativo. Riuscire a effettuare un simile lavoro su una piattaforma embedded di queste dimensioni sarebbe davvero un bel traguardo.

Si potrebbero inoltre estendere le funzionalità del client tramite le librerie di OpenCV per il riconoscimento automatico di volti, forme ed oggetti, e sfruttare il protocollo TCP per azionare differenti comportamenti sulla board ogni volta che si verifica un dato evento.

Capitolo 7 - Conclusioni

Il fine di questo lavoro di tesi sperimentale è stato quello, partendo da un sistema per l'elaborazione e la trasmissione di immagini precedentemente sviluppato da altri colleghi, di aumentarne l'efficienza ed estenderne le funzionalità.

Il sistema in oggetto è composto da una scheda elettronica programmabile *ZedBoard*, prodotta da *Xilinx*, che è in grado di acquisire ed elaborare tramite FPGA le immagini catturate dal gruppo di sensori ad essa collegati *OmniVision OV7670*.

Attraverso lo studio e l'analisi del codice esistente, e tramite i più moderni ambienti di sviluppo, ho potuto migliorare la leggibilità del codice stesso e la sua futura estensibilità.

Sintetizzando, le operazioni principali su cui ho concentrato il mio lavoro sono state:

- Riorganizzazione e modularizzazione del progetto esistente:

  Queste modifiche permetteranno ai futuri sviluppatori una più immediata comprensione del funzionamento del software, e la possibile esclusione di singoli moduli senza bisogno di stravolgere il codice sorgente riducendo notevolmente il rischio di errore.

- Implementazione di comunicazioni sicure mediante il protocollo TCP:

  L'introduzione di questo nuovo metodo di comunicazione tra board e client consentirà di controllare alcune funzionalità della board da remoto senza bisogno di agire fisicamente sull'hardware. La scelta del protocollo utilizzato è stata dettata dalla sua affidabilità e sicurezza poiché è estremamente importante avere garanzia che le informazioni trasmesse arrivino a destinazione.

- Scrittura del client per la ricezione delle immagini a colori:

  Attraverso l'inserimento di nuove funzionalità sulla ZedBoard è stato possibile inviare e ricevere sul client anche i frame contenenti le componenti di luminanza e crominanza utili alla visualizzazione di un flusso di immagini nello spazio di colore RGB. Conseguentemente a questa operazione l'utilizzo della banda risulta quasi triplicato comportando un lieve peggioramento della corretta trasmissione e ricezione delle immagini. Vi sarà quindi un margine di miglioramento futuro per correggere il numero di frame (datagrammi) persi.

Bibliografia

[1] www.zedboard.org

[2] www.xilinx.com

[3] www.doxygen.org

[4] www.opencv.org

[5] www.xilinx.com/products/design-tools/vivado.html

[6] www.xilinx.com/products/design-tools/embedded-software/sdk.html

[7] www.eclipse.org

[8] www.codeblocks.org

[9] www.mingw.org

[10] www.voti.nl/docs/OV7670.pdf

[11] www.wireshark.org

[12] www.wikipedia.org/wiki/Javadoc

[13] www.microsoft.com

[14] www.msdn.microsoft.com/en-us/library/windows/desktop/ms738545(v=vs.85).aspx

[15] www.wikipedia.org/wiki/Transmission_Control_Protocol

[16] Riccardo Albertazzi, “Sistema di visione stereo su architettura Zynq”, Università di Bologna, tesi di laurea A.A. 2015/16

[17] Simone Mingareli, “Streaming di immagini via ethernet con Zynq con sistemi operativi Standalone e Linux”, Università di Bologna, tesi di laurea A.A. 2015/16

[18] www.lwip.wikia.com/wiki/LwIP_Wiki

Ringraziamenti

Fin da quando ero bambino ho sempre pensato che da “grande” avrei fatto il medico seguendo le orme di papà. Ho iniziato la mia carriera universitaria affrontando il test di ingesso per la Facoltà di Medicina e Chirurgia che non ho, oggi penso fortunatamente, superato.

Dopo un incerto inizio alla Facoltà di Scienze Naturali, ho capito grazie all’aiuto e al conforto delle parole di mamma, che né medicina né i mitocondri sarebbero stati la mia strada. Mi è sempre piaciuta l’informatica perciò alla domanda di mia madre: “Cosa vuoi per la tua vita?” non è stato difficile risponderle che sarei voluto diventare un ingegnere informatico.

Ringrazio anzitutto il professor Stefano Mattoccia, senza la sua guida sapiente questa tesi non esisterebbe.

Ringrazio i miei numi tutelari, Mimma e Pietro, che mi hanno supportato e sopportato nei primi difficili passi con la loro dedizione ed attenzione.

Un ringraziamento particolare va ai colleghi del mio gruppo di studi, Luca, Giuseppe, Isidoro e Zakaria: grazie alla loro disponibilità, gentilezza e alla sincera amicizia mi è stato possibile concludere questo mio percorso portando con me splendidi ricordi delle notti passate insieme a programmare, studiare e giocare a LoL.

Ringrazio inoltre i miei amici d’infanzia Federico, Marco ed Alessandro che mi hanno sempre incoraggiato e sostenuto in tutte le difficili scelte e situazioni che ho dovuto affrontare per arrivare alla conclusione dei miei studi.

Vorrei infine ringraziare le persone a me più care: i miei genitori, per avermi permesso di affrontare gli studi senza farmi mancare nulla, mia sorella ed Irene per la solidarietà e gli incoraggiamenti.