ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DISI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA E TELECOMUNICAZIONI

TESI DI LAUREA

Mapping of computer vision modules on FPGA and images transmision via Ethernet

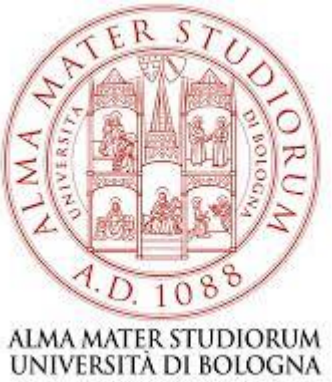
AUTORI:

Diego Garay Esterán

David Fernández Villa

RELATORE:

Stefano Mattoccia



INDICE

1. Introduzione

In queste mondo attuale, i sistemi de visione hanno una maggiore importanza e anche una larga diffusione. Sia per applicazioni come videosorveglianza, ricostruzione di una scena 3D oppure altri applicazioni diverse nelle quali possono esse sfrutatti.

Il nostro lavoro di tesi si trova dentro di un progetto molto amplio e complesso su il ottenimento dei immagini, la loro gestione e ricostruzione della scena, e il loro analisi con diversi obiettivi, tra cui, il obiettivo principale è quelo di individuare gli ostacoli presenti.

Per raggiungere il nostro obiettivo, in anzi tutto, dobbia fare il design del hardware di cui abbiamo bisogno, tramite Vivado, facendo attenzione a lo spazio d'Indirizzamento e le conesioni giusti per il corretto funzionamento. Dobbiamo dire, che per il test iniziale abbiamo usato un Test Pattern Generator in sostituzione della telecamera.

Dopo di questo, comiciamo con la programazione del driver che gestisci il VDMA, prima en *baremetal* e poi, nel Sistema Operativo Linaro basato su Linux. Anche se questo sembra banale, comporta un largo studio sull'utilizzo e funzionamento del sudetto driver.

Una volta che sia fatto, mancherebbe la installazione del Sistema Operativo Linaro, che sará approfondito nel argomento corrispondente, per il corretto funzionamento del driver e il *server* di invio degli immagini.

Finalmente, abbiamo fatto un piccolo *server* che fa la estrazione dei frame e il invio tramite *ethernet* al host corrispondente per la gestione e il analisi degli immagini ottenuti.

2. La ZedBoard: ZYNQ-7000

La ZedBoard è una scheda di valutazione e sviluppo basata sulla Xilinx ZynqTM -7000 All Programmabile SoC (AP SoC) . La combinazione di un sistema di elaborazione dual Corex - A9 (PS) con 85.000 (PL), le cellule della serie -7 logica programmabile , il Zynq -7000 AP SoC possono essere oggetto di ampio utilizzo in molte applicazioni . Robusto mix del ZedBoard di periferiche on -board e di espansione funzionalità ne fanno una piattaforma ideale sia per i principianti ed esperti designer.

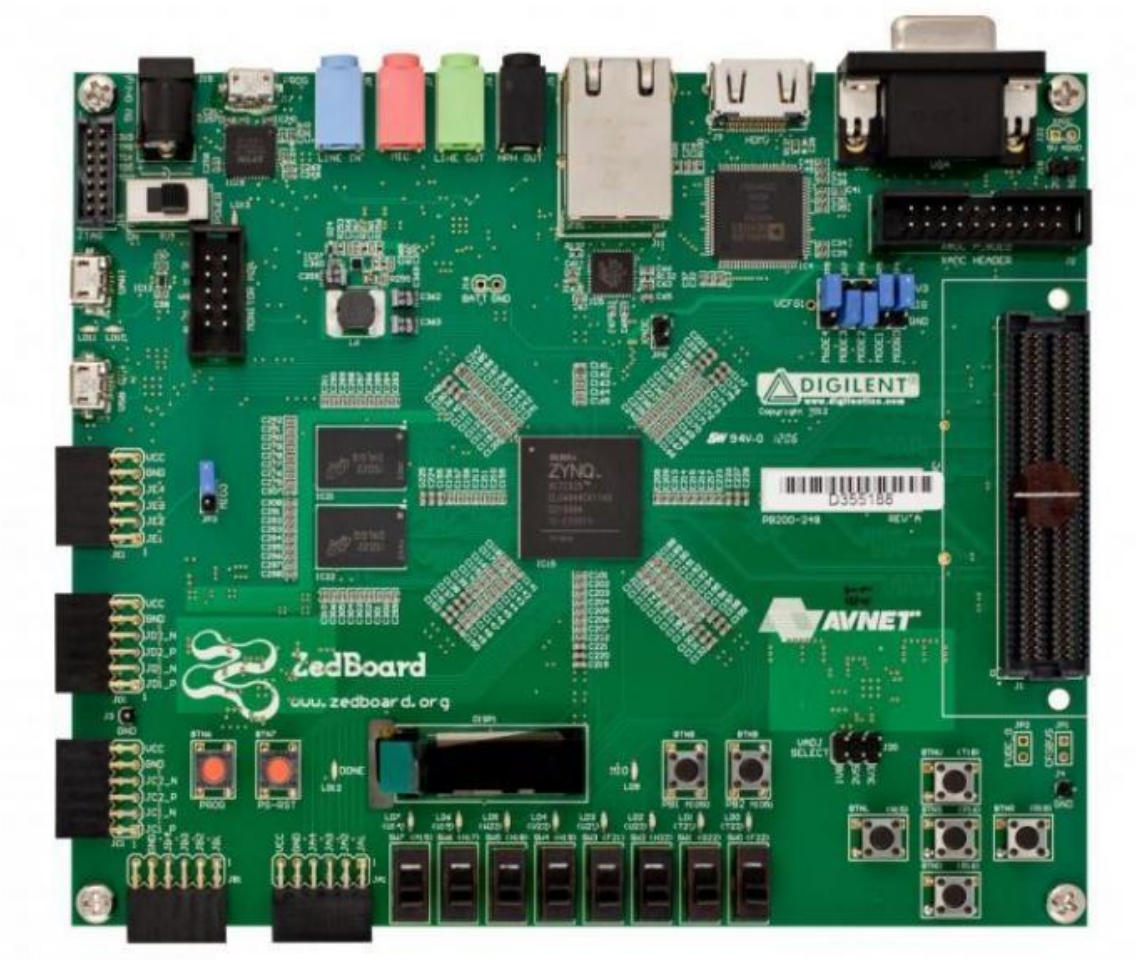


Figure 2-1: ZedBoard Zynq 7000

PS o Processing System → È la parte non programmabile, dove il software è in esecuzione. Composto da un processore ARM dual - core Cortex - A9 che governa il sistema.

PL o Programmable Logic → È la parte programmabile, dove si farà l'implementazione di dispositivi che sarano comunicati con il PS tramite un'interfaccia AXI .

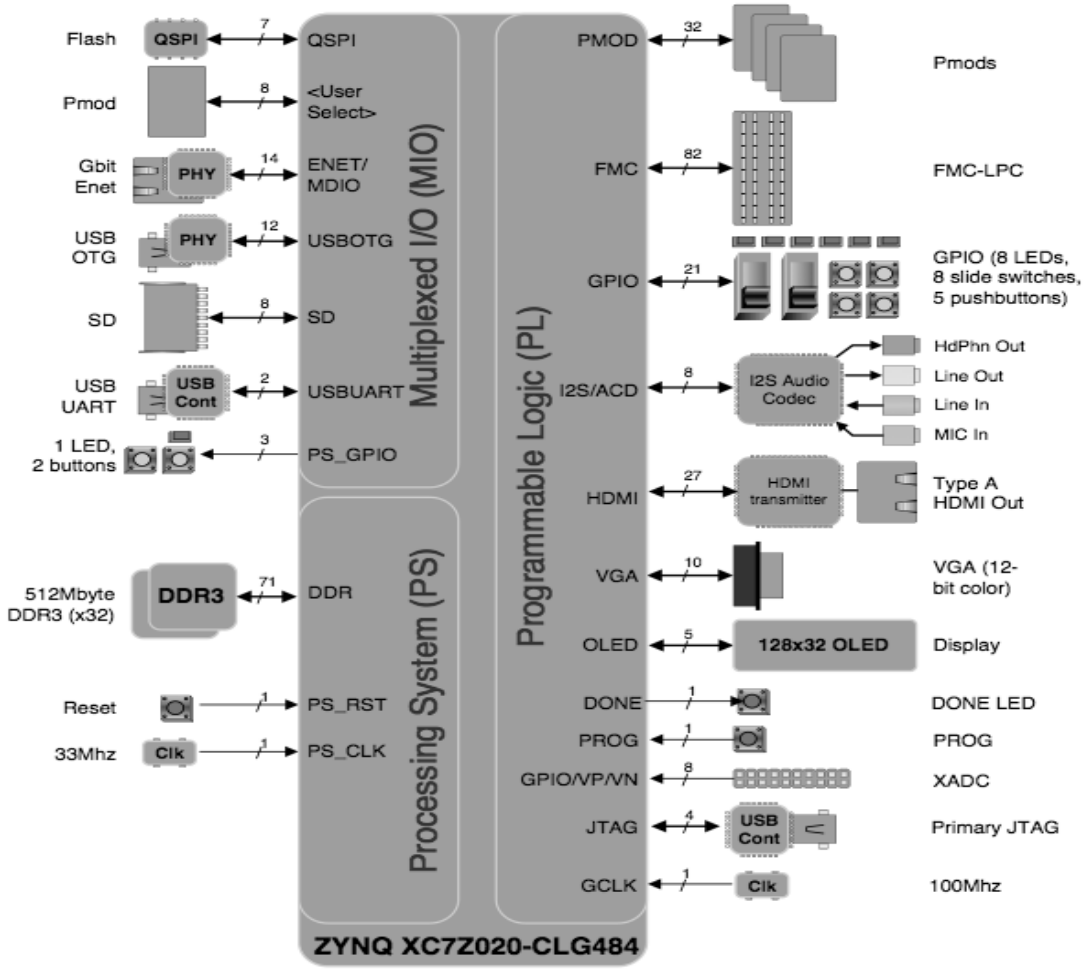


Figure 2-2:ZedBoard Layout spiegato prima

Come la scheda ha inclusa una APU, si puo dire che contiene tutto il necesario per l'impiego di un sistema operativo. Permette creare un progretto Linux, Android, Windows o per altri OS. Il più utilizzato è sicuramente Linux.

La presenza di diversi connetori di espansione fanno la scheda molto flessibile. Cosi l'elaborazione e la logica programmabile di I/O sarà più facile e adattabile alle esigenze dell'utente.

3. Strumenti di sviluppo

3.1.VIVADO

E ' il principale strumento per la progettazione di piattaforma hardware. Completamente preparato per piattaforme Zynq , questo ambiente permette entrambi la configurazione e personalizzazione del sistema Programma (PS) di quest come l'aggiunta di piattaforme e interconnessione dei blocchi logici o IP (Intellectual Proprietà) nella FPGA o PL , tutto questo in un ambiente che integra le due sezioni. Offerte soluzioni per l' automazione dei processi e ha un’altri vantaggi che permettono semplificare e accelerare notevolmente il tempo di sviluppo di un disegno mentre offrendo l'integrazione con altri strumenti che rendono questa suite .

3.2.IP-Cores

Proprietà intellettuale (IP) sono elementi chiave della piattaforme di progettazione Xilinx mirati . Dispositivi e strumenti Xilinx FPGA sono state progettate per la facile creazione di Plug -and-Play IP. Servono per potere fare il disegno (la logiche della aplicazione che il utente vuo disegnare). Questo strumento e molto imporante perchè offerte numerosi opzioni per fare diversi disegni.

3.3.SDK

Il Software Development Kit è stato realizzato basandosi su Ecplise ed offre il supporto, al progettista, per lo sviluppo e il debug di codice C o C++ per la piattaforma hardware specifica; offre la possibilità di realizzare applicazioni bare-metal (senza sistema operativo) e applicazioni Linux based, sia per sistemi monoprocessore sia multiprocessore. L'obiettivo di questo tool è quello di compilare il codice scritto dal progettista, includendo i driver delle periferiche presenti nel sistema ed eventuali altre librerie, al fine di creare un file in formato ELF (Executable Linked Format), direttamente eseguibile da parte del

processore della piattaforma. Il processo di generazione del file ELF da parte di SDK sfrutta il tool Xilinx LibGen, oltre al compilatore e al linker GCC. LibGen impiega i file MHS e MMS (creati tramite XPS), i driver degli IP-Core, eventuali librerie e sistema operativo per la creazione del Board Support Package (BSP), utilizzabile per lo sviluppo del software. SDK, è in grado di creare il file ELF; ciò è possibile solo dopo aver generato, tramite wizard, l'opportuno Linker Script.

Serve pure per generare i file First Stage Boot Loader (FSBL), BOOT.bin e devicetree.dts che sono necessari per fare la corretta configurazione, installazione e boot di qualsiasi sistema operativo Linux di cui si approfondirà nell argomento su Linux.

4. Progetto Hardware

Questa e la prima parte reale dil progetto. Il lavoro in questa parte cerca trovare un meccanismo per ricevere le immagini provenienti da una camara e, ultilizando un disegno vivado, memorizarrle nella memoria DDR3 dealla ZedBoard.

Prima di tutto si deve fare un nuovo progretto in Xilinx Vivado:

1- Run Vivado 2.014.4 (o la versione disponibile) del software .
2- Per creare un nuovo progetto , fare clic sulla finestra *Creat new project*, o il menu *File- > New project*...
3- Premere next e lasciare tutto invariato fino alla schermata *default* part dove dobbiamo selezionare la zedboard come target per lo sviluppo. La opzione *Create project subdirectory* farà un elenco con il nome del progetto nel percorso specificato, quindi è utile per separare e organizzare i progetti intrapresi .
4- Nella forma *Project Type* troverete una serie di opzioni tra cui scegliere il tipo di progretto. *RTL Protject* sarà selezionato perché vogliamo modificare il nostro disegno e realizare l'analisis, la sintesi e l'attuazione. Fare click in Next.
5- Come stiamo usando una scheda di sviluppo ZedBoard, selezioniamo Boards nella casella *Spedify* e segnare la scela la opzione *ZedBoard Zynq Evalutaion and Development Kit* nel riquadro inferiore. Fare click in Next.
6- Arriveriamo all'ultima finestra della procedura guidata. Fare click su *Finish* per cerare il progretto e comenziare a lavorare.

4.1.Block Design

Per creare un nuovo disegno basato in blochi Hardware selezionare *Create Block Design* nel bar *Flow Navigator -> IP Integrator.* Selezionare un nome per la progrettazione. Nella scheda *Diagram* mostrata possiamo aggiungere blocchi hardware predefiniti per catalogo o blochi personalizzati creati in precendenza. Clicando due volte sulle module IP se puo modificare la sua configurazione interna.

Questo sarà il Block Design per realizzare l'obiettivo:

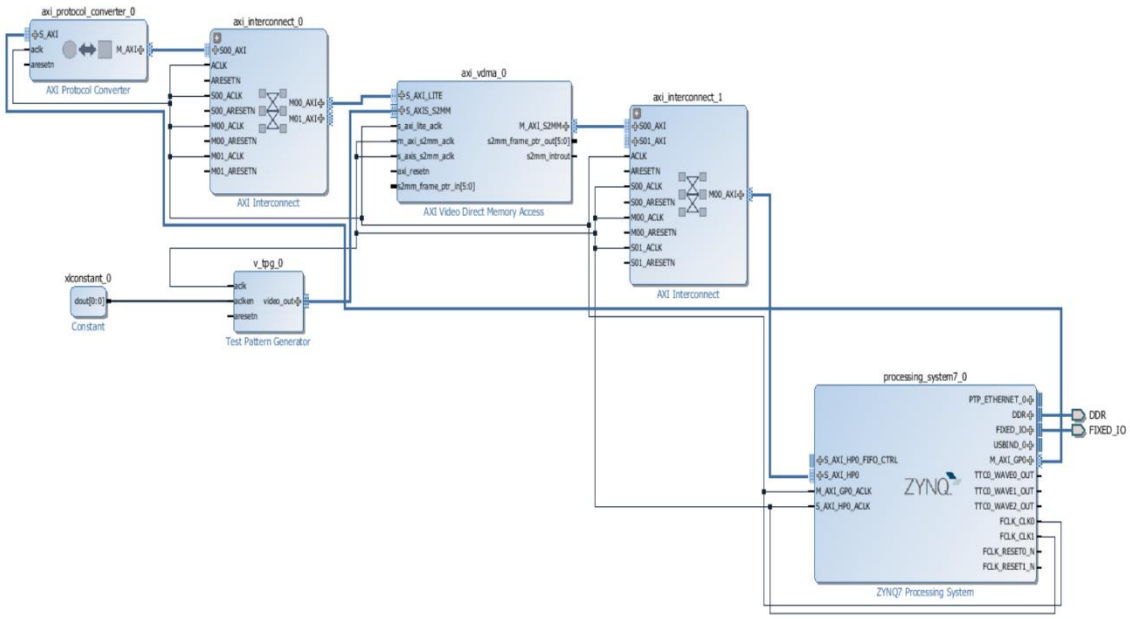


Figure 4-1:Block Design

Adesso si spiega ogni IP e perchè vengono ultilizzati:

- ZYNQ:

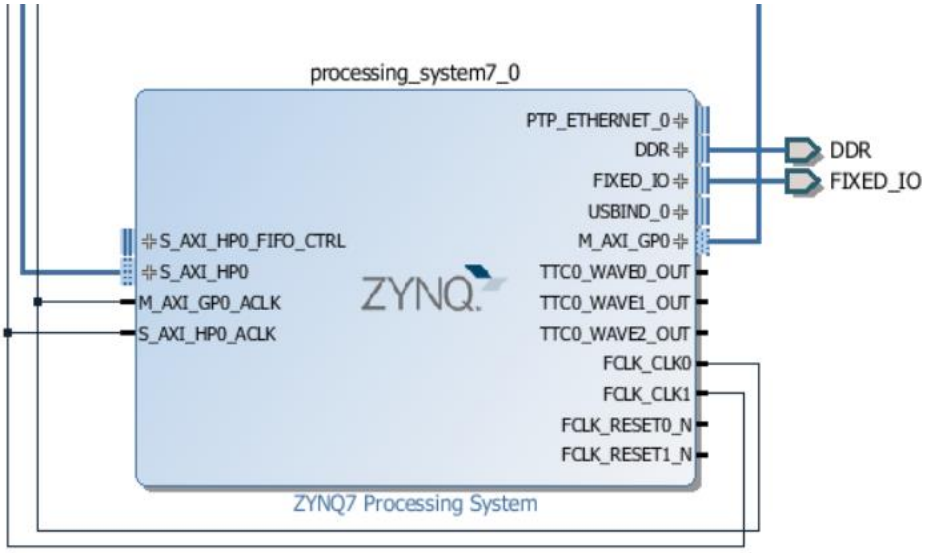


Figure 4-2:Blocco principale del disegno Zynq

Il Zynq è il modulo principale, è il centro di computazione e gestione la memoria. Nel questo caso lo abbiamo usato per essere in grado di programmare il VDMA. Si può observare che ci sono due reset. Questo e perchè il protocolo AXI dici que le conessione AXI4-LITE slaves sono comunicati a una velocita di 100MHz nel bus pero i segniali di video sono trasnferiti a una frequenza di 150MHz .

Ingressi e le uscite più importanti:

- S_AXI_HP0: Axi slave (input) connessione ad alta performance in cui ci colleghiamo la VDMA e attraverso la quale l'accesso alla memoria per posizionare le immagini.
- M_AXI_GP0: Axi Master (output) che useremo per configurare il VDMA.
- FCLK_CLK0 e FCLK_RESET0_N: Signale di clock e reset per tutti i moduli.

Ultiliza il protocolo AXI che sarà spiegato dopo.

- AXI INTERCONNECT:

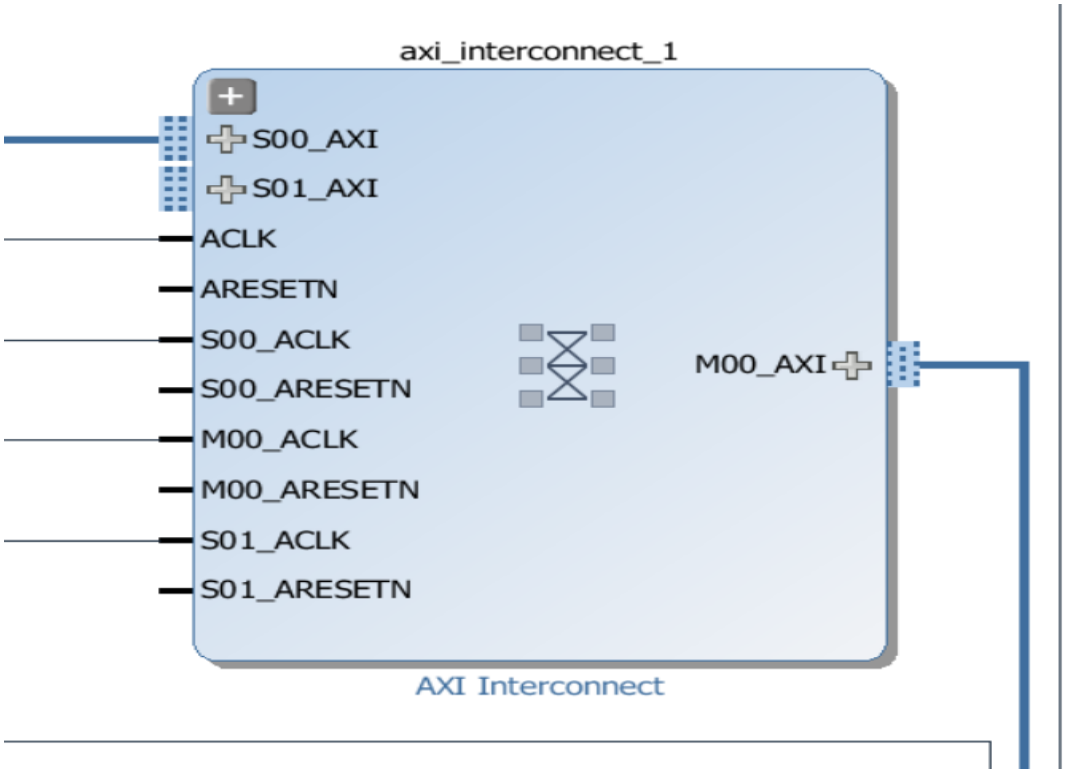


Figure 4-3:Axi Interconnect

Interconnessione tra lo Zynq e il VDMA per potere fare la configurazione e programazione dal modulo. Al l'ingresso riceve, dal Zynq un collegamento AXI master con le impostazioni VDMA (entrata M00_AXI) e esci la connessione necessaria per la configurazione del DMA. Pertanto la sua seria funzione adattatore. Questo blocco anche mantiene le due reset.

- AXI VDMA:

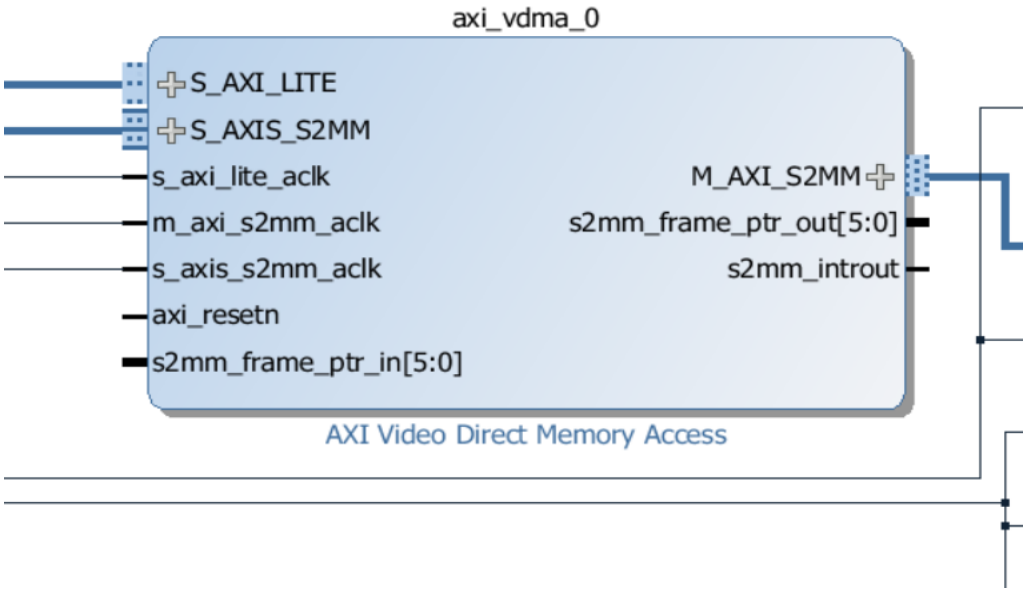


Figure 4-4:AXI VDMA il blocco più importante per il trasnfiremento del video

Dopo lo Zynq, è il modulo più importante. Fai la funzione primaria del nostro modulo, dal momento che la sua funzione è quella di ricevere immagini o fotogrammi, dalla fotocamera, memorizzarli nel buffer interno e quando è necessario inviare loro a memoria. Quest o è il modulo che abbiamo dovuto programmare.

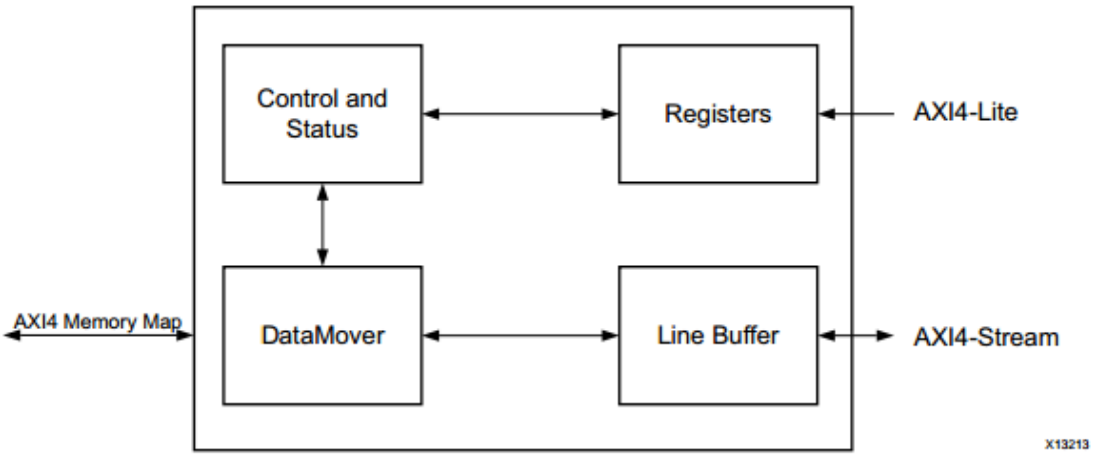


Figure 4-5:AXI VDMA Block Design.

Di questo blocco è programmato il driver per potete fare un test e finalmente con la FPGA fare il trasnferimento d'immagini.

- Test Pattern Generator:

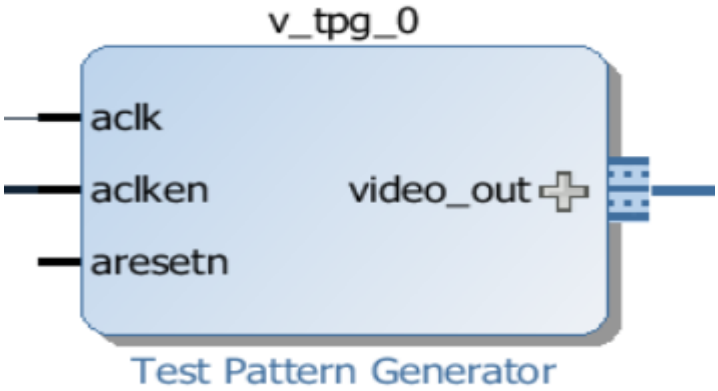


Figure 4-6:Test Pattern Generator importante per potere fare test

Per effettuare il test, utilizzare questo modulo come una simulazione del la telecamera poiché la sua funzione è quella di inviare immagini VDMA per l'elaborazione. In questo caso abbiamo scelto le immagini monocromatiche della stessa risoluzione fotocamera (640x480) e frequenza. La sua configurazione è necessaria per la simulazione e la sperimentazione di altri moduli.

Questo blocco deve essere configurato cosi:

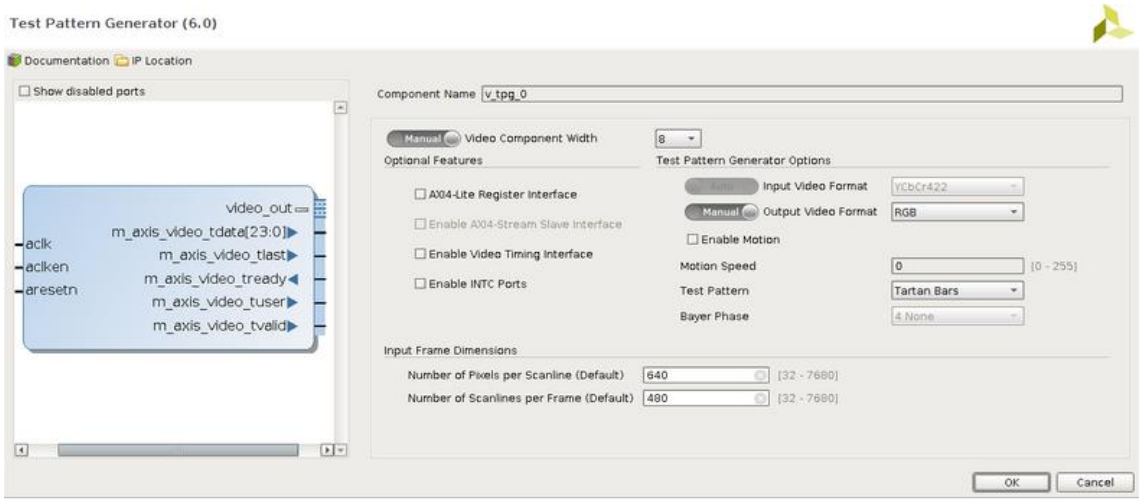


Figure 4-7:Configurazoine del Test Pattern Generator

Con questa configurazione, sappiamo come sarà l'uscit, quindi sapiamo che cosa si dovrebbe vedere dopo (per questo è importante la configurazione del Pattern). Sarebbe qualcosa cosi:

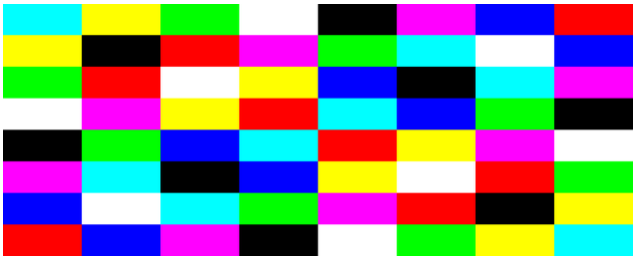


Figure 4-8:Imagine generata dal pattern.

4.2.Protocollo AXI

Le specificacione AXI descrivono un'interfaccia tra un singolo master AXI e una singola AXI schiava, raprresentando core IP che scambiano informazione tra loro. Le due possono essere collegati tra loro grazie a una struttura chiamata Interconnect blocco. Il AXI Interconnect Xilinx IP contiene interfacce master e slave AXI supportati, e può essere ultilizzato per operazioni di percorso tra uno o più master e slave AXI.

Le due interfaccia AXI4 e AXI4 – Lite hanno cinque canali diversi:

1- Leggi Canale Indirizzo
2- Scrivere Canale Indirizzo
3- Lettura dati Canale
4- Scrivere dati Canale
5- Scrivere Risposta descrittto Channel logs.

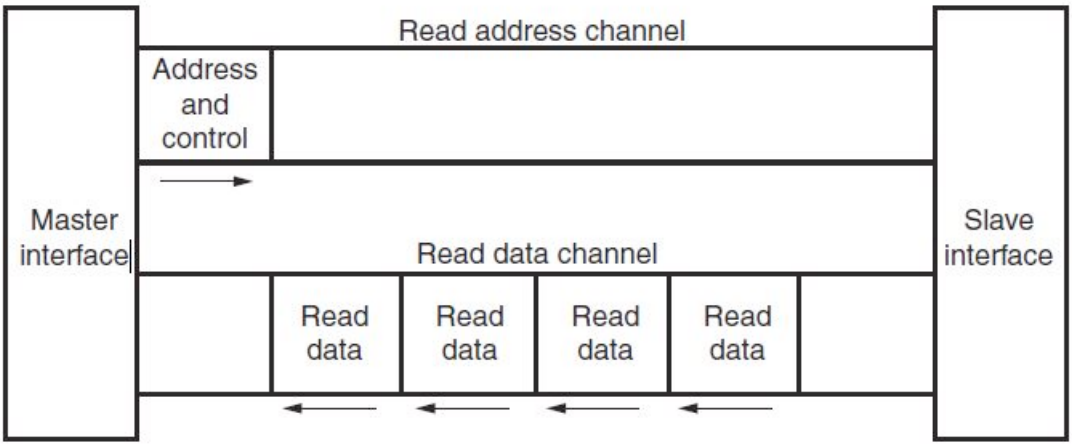


Figure 4-9: Channel Architecture of Reads

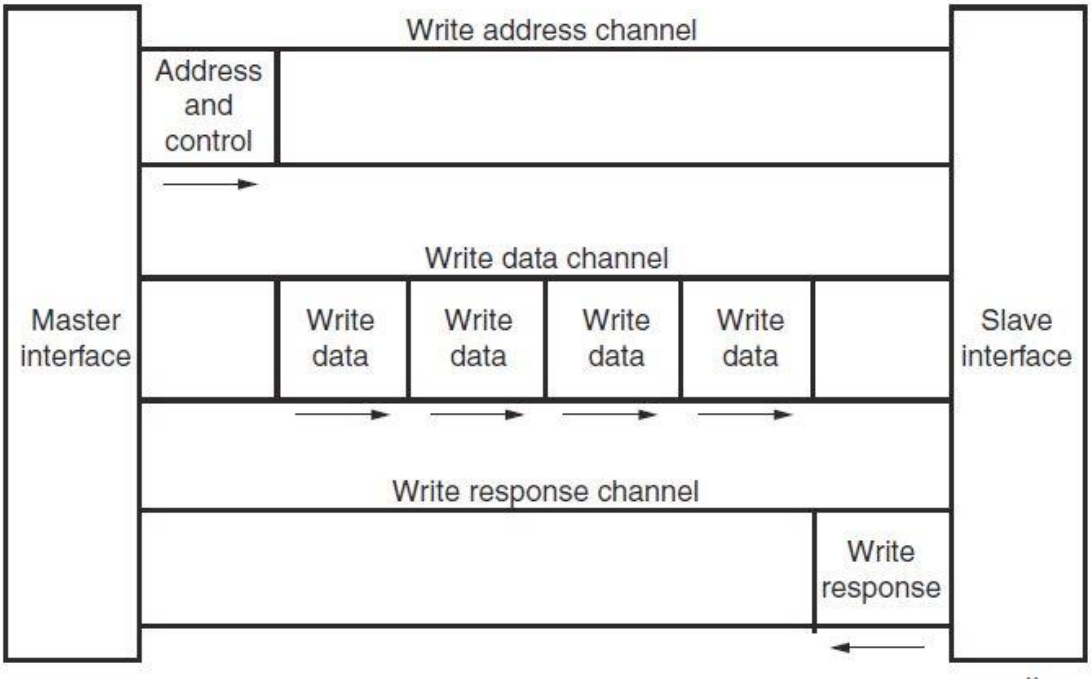


Figure 4-10: Channel Architecture of Writes

Come mostrato nelle figure precedenti , AXI4 fornisce dati separati e collegamenti dell'indirizzo per lettura e scrittura , che consente contemporaneamente , il trasferimento di dati bidirezionale . AXI4 richiede un singolo indirizzo e poi scoppia fino a 256 parole di dati . Il protocollo AXI4 descrive un varietà di opzioni che permettono sistemi AXI4 compatibile per raggiungere molto elevato throughput dei dati . Alcune di queste caratteristiche , oltre a scoppio , sono :

upsize dati e ridimensionamento , multipla gli indirizzi in sospeso, e out-of -order l'elaborazione delle transazioni .

A livello hardware, AXI4 permette un orologio diverso per ogni coppia AXI master-slave . In aggiunta, il protocollo AXI permette l'inserimento di fette di registro (spesso chiamati stadi della pipeline)per facilitare temporizzazione chiusura .

AXI4 - Lite è simile a AXI4 con alcune eccezioni, la più importante delle quali è che scoppio, non è supportato. Il capitolo AXI4 -Lite della ARM AMBA AXI protocollo v2.0 Specifica descrive il protocollo AXI4 - Lite nel dettaglio.

Il protocollo AXI4 -Stream definisce un unico canale per la trasmissione dei dati in streaming . il Canale AXI4 -Stream è modellato dopo il canale dati di scrittura del AXI4. A differenza AXI4, Interfacce

AXI4 -Stream possono scoppiare una quantità illimitata di dati. Ci sono aggiuntivi , funzionalità opzionali descritte nella specifica v1.0 AMBA4 AXI4 -Stream Protocol. La specifica descrive come interfacce AXI4 -Stream - compliant possono essere divisi , fuse ,interlacciata, upsize, e ridimensionato. A differenza AXI4, trasferimenti AXI4 - Stream non può essere riordinati .

4.3.AXI4-Stream

Questo protocollo risulta particolarmente adatto per applicazioni nelle quali è richiesto lo scambio di una notevole quantità di dati e in cui non è presente, o non è richiesto, il concetto di indirizzamento. Ogni connessione AXI-Stream (AXI4S) è point-to-point, monodirezionale e di tipo Handshake. Dato l'alto numero di ambiti in cui esso è utilizzabile, i segnali del protocollo assumono significato diverso in base al contesto nel quale sono utilizzati; riportiamo, di seguito, i principali segnali del protocollo e il significato assunto nell'ambito di applicazioni video:

Function	Width	AXIS Signal Name	Video Specific Name
Video Data	8, 16, 24, 32,40, 48,56,64	tdata	DATA
Valid	1	tvalid	VALID
Ready	1	tready	READY
Start Of Frame	1	tuser	SOF
End Of Line	1	tlast	EOL
Clock	1	aclk	ACLK

Tab 4-1: AXI4-Stream Video Protocol Signals

Prima di dare una descrizione dettagliata di tali segnali, e di come essi siano utilizzati, occorre specificare che AXI-Stream prevede il trasferimento dei soli pixel effettivamente appartenenti all'immagine.

L'unico segnale controllato dal dispositivo Slave è il Ready ed è utilizzato per indicare la disponibilità a ricevere dati; tutti gli altri sono sotto la responsabilità del Master. Il Master utilizza il segnale Valid per indicare il trasferimento di un pixel appartenente all'immagine, il cui valore è trasmesso attraverso i segnali Video Data.

Il segnale Start Of Frame è attivato finché non ne è confermata la ricezione, durante la trasmissione del primo pixel di un frame; il segnale End Of Line indica, invece, la fine di una linea, ed è attivato durante la trasmissione dell'ultimo pixel della stessa. Ecco un esempio di utilizzo di questo protocollo per la trasmissione di immagini aventi N pixel per riga.

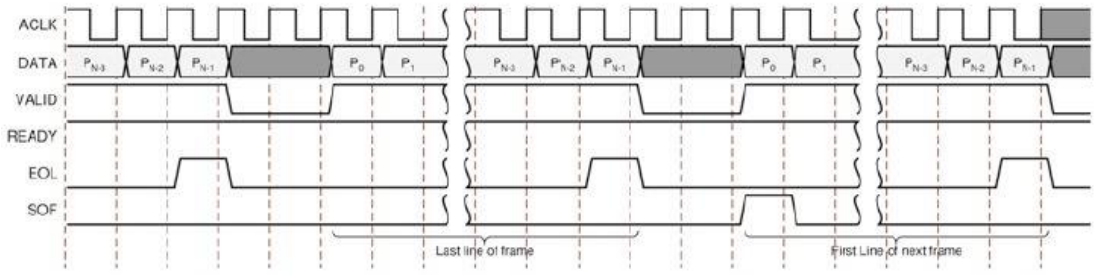


Figure 4-11: Esempio trasferimento flusso video con Axi-Stream

4.4.AXI Interconnect Core

Questo IP-Core offre la possibilità di collegare fino a 16 Master AXI4 (o AXI4-Lite), con un massimo di 16 Slave [11]; i dispositivi, connessi tramite questa infrastruttura di comunicazione, possono avere interfacce con parallelismo diverso e clock differenziati, rendendo l'AXI molto flessibile e potente. Se lo si utilizza per connettere semplicemente un Master ed uno Slave, aventi lo stesso parallelismo e lo stesso clock, questo IP-Core si comporterà come una serie di fili tra essi, senza andare ad occupare nessuna risorsa della PL (Fig.4-12); aumentando le funzionalità richieste, la fase di sintesi richiederà maggiori risorse e saranno introdotti dei ritardi nel canale di comunicazione, causati dalle operazioni effettuate per rendere compatibili tutti i dispositivi interconnessi (Fig.4-13).

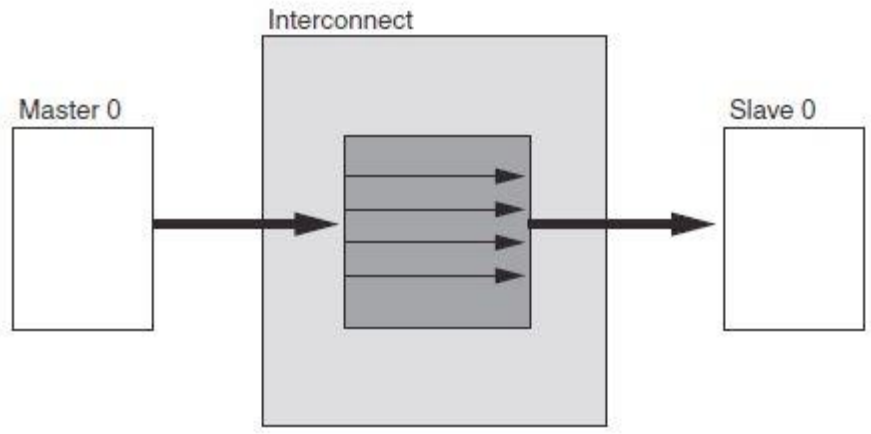


Figure 4-12:1-1 Pass-through AXI Interconnect

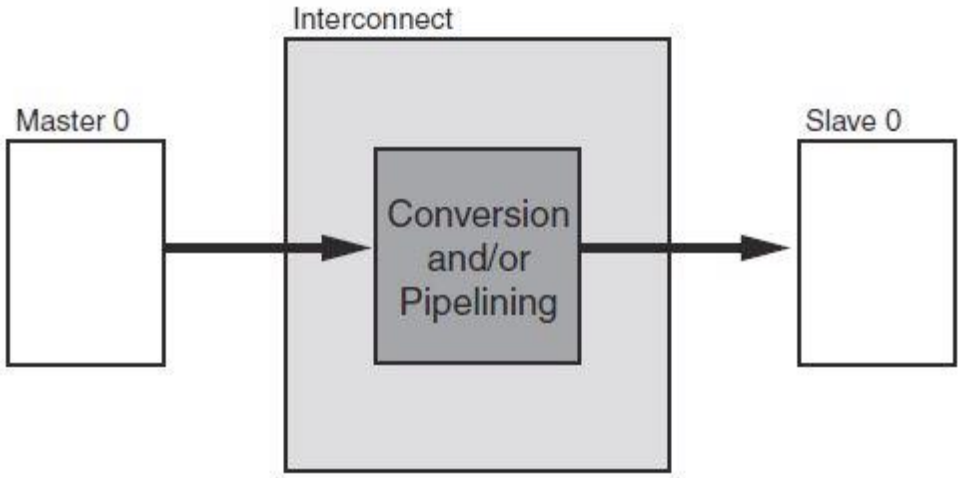


Figure 4-13:1-1 AXI Interconnect con conversioni

Un'altra caratteristica importante è che, tramite lo stesso blocco Interconnect, è possibile far comunicare dispositivi sia con interfaccia AXI4 sia con AXI4-Lite. Si mostra ora una visione ad alto livello della struttura di questo IP-Core:

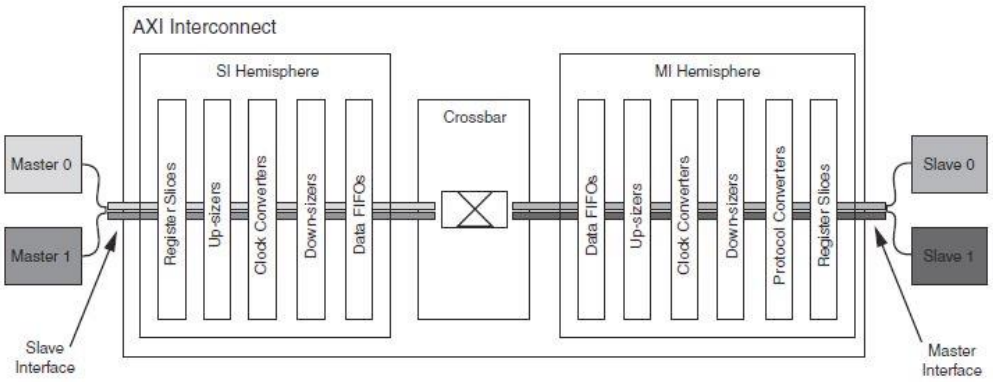


Figure 4-14:IP-Core AXI Interconnect

L'AXI Interconnect è costituito da una Slave Interface (SI), una Master Interface (MI) e dall'unità funzionale che crea un percorso tra esse (crossbar).

Le funzionalità aggiuntive, descritte in precedenza, sono realizzate dai componenti presenti tra la singola interfaccia e la crossbar stessa. Nel caso in cui sono presenti più master, la priorità tra essi è decisa da un arbitro, configurabile con priorità fisse o per lavorare in modo Round Robin. Un esempio nel quale è necessario l'intervento dell'arbitro, è il seguente:

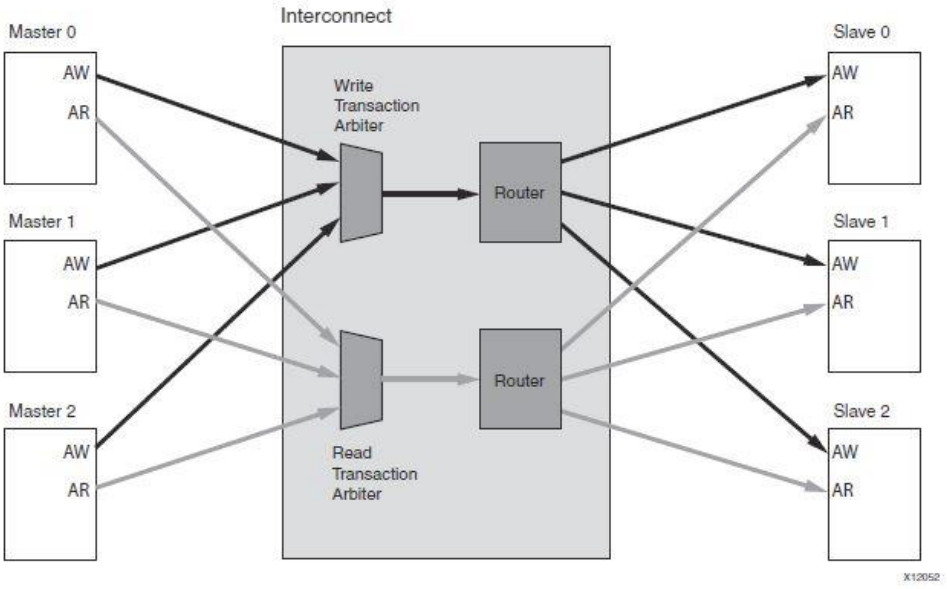


Figure 4-15: Shared Write and Read Address Arbitration

4.5.Test Pattern Generator

Questo IP-Core genera Test Patterns di Video per lo sviluppo, la valutazione e il debugging di progetti basati su l'utilizzo di segnali di video. Il suddetto IP-Core fornisce una grandisima varietà di pattern con diversi forme, colore, resoluzione e frecuenza. Il core si può collegare ad una interfaccia di video AXI4-Stream che permette al usuario recivere devirse segnali di video in. Anche si può programmare in runtime tramite una interfaccia AXI4-Lite.

Il TPG è semplice da usare. Per fare la configurazione, si utilizza la interfaccia di configurazione data per il programa Vivado e poi collegare correttamente il port di Video-out del TPG con il port di Video-in del nostro IP-Core che gestiona il flusso di video.

Le dimensione dei dati di video generati per il TPG sono multipli di 8 bits, però si può usare altre dimensione non multipli di 8 bits dopo di fare un padding necesario, come si mostra in figura 4-16, e che non influise in le dimensione del core.

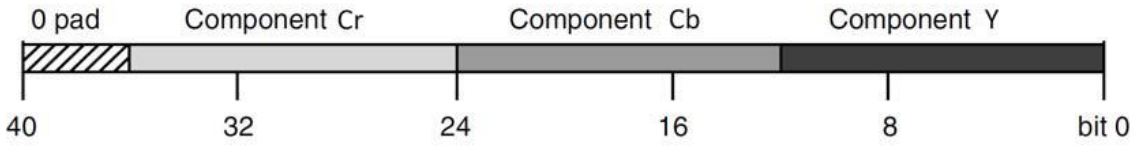


Figure 4-16: Video Data YCbCr Padding

4.6.AXI VDMA

Questo IP-Core permette il trasferimento di dati, fornendo una grande larghezza di banda, tra la memoria e un dispositivo che presenta l'interfaccia AXI-Stream [12]. Questo componente è costituito da due canali:

- CANALE DI SCRITTURA (S2MM): sfruttando questo canale è in grado di ricevere immagini attraverso l'AXIS e, assumendo il ruolo di Master AXI, bufferizza i frame in memoria.

- CANALE DI LETTURA (MM2S): si tratta dell'operazione duale, intendendo con questo termine l'estrazione dalla memoria di un frame, il quale è inviato successivamente ad uno slave con l'interfaccia AXIS.

Questo IP-Core assume, quindi, il ruolo di Master e quello di Slave, sia per l'AXI sia per l'AXI-Stream; presenta, inoltre, un'interfaccia Slave AXI-Lite, da utilizzare per la sua programmazione. Nel caso dello Zynq entrambi i canali sfruttano, per accedere alla memoria, una delle porte High-Performance presenti tra PS e PL, poiché, come detto in precedenza, sono quelle che garantiscono il maggior throughput. Ogni canale è attivabile (e configurabile) in maniera totalmente indipendente dall'altro. In figura 4-17 è riportato un diagramma a blocchi del VDMA.

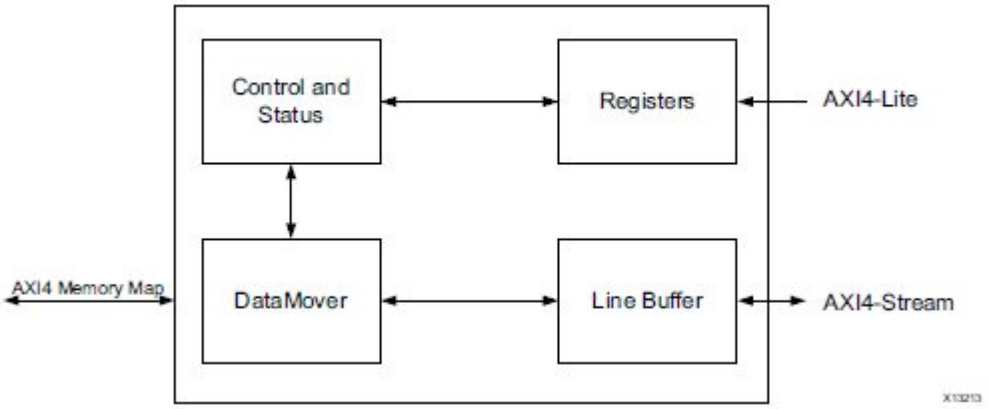


Figure 4-17:Diagramma a blocchi del VDMA

Dopo aver programmato i registri, attraverso l'AXI-Lite, il blocco denominato Control and Status genera i comandi appropriati per il DataMover, il quale inizia la lettura o la scrittura tramite l'interfaccia AXI4. Come si può notare, è presente, inoltre, un Line Buffer nel quale sono depositati i valori dei pixel, prima che essi siano trasferiti.

Il VDMA offre la possibilità di realizzare un frame buffer circolare, memorizzando un numero configurabile di frame (al massimo 4-19).

Per il seguente lavoro di tesi si è sfruttato solamente il canale di scrittura verso la memoria, l'estrazione dei frame è stata poi effettuata tramite il

processore. Mostriamo ora (figura 4-18) un esempio, con relative temporizzazioni, del funzionamento di questo canale.

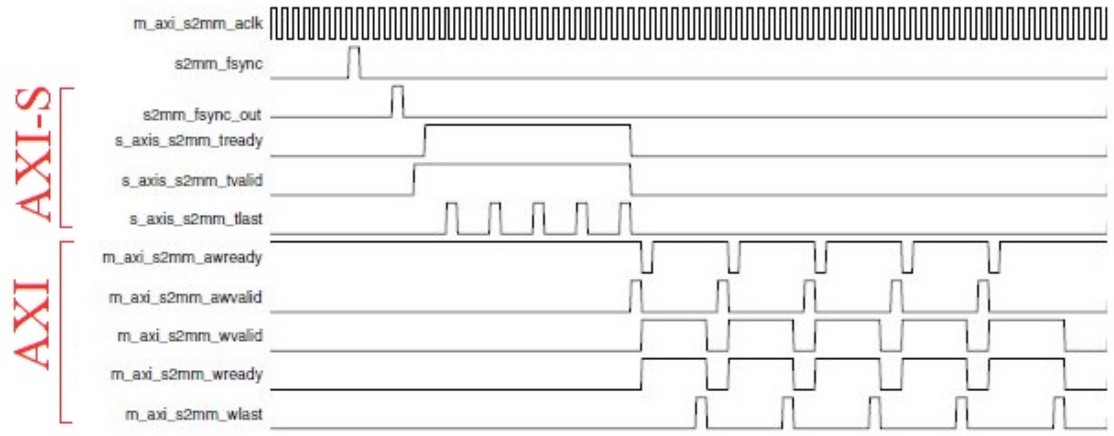


Figure 4-18:Esempio VDMA S2MM Channel

In tal caso, un'immagine è costituita da 5 linee, ognuna di 16 byte. Il VDMA, dopo aver ricevuto il segnale fsync, si rende disponibile a ricevere dati sull'interfaccia AXIS, asserendo il segnale tready; lo streaming di dati è memorizzato nel line buffer e, successivamente, trasferito in memoriasfruttando l'interfaccia AXI.

Il VDMA offre anche la possibilità di sincronizzare il canale di scrittura con quello di lettura, al fine di evitare l'accesso contemporaneo allo stesso frame. Parte di questo meccanismo è stato sfruttato per evitare che il processore prelevi il frame che il VDMA sta scrivendo, operazione assolutamente da non compiere.

Xilinx offre, per la programmazione di questo IP-Core, i relativi driver, sia per Linux sia per applicazioni di tipo Bare-Metal. L'esistenza di questi driver evita, al programmatore, di dover configurare il componente tramite la lettura e scrittura diretta dei registri, operazione che risulterebbe molto delicata e complessa.

Per il raggiungimento dell'obiettivo di questo lavoro di tesi si vogliono gestire dei flussi d'immagini provenienti da sensori e servirà quindi aggiungere, a monte del VDMA, un componente in grado di presentare tali immagini in modo compatibile con l'interfaccia AXI-Stream; tale componente esiste tra quelli offerti da Xilinx, ed è il Video in to Axi-Stream, illustrato nella prossima sezione.

In figura 4-19 viene mostrato il VDMA, e le relative interfacce:

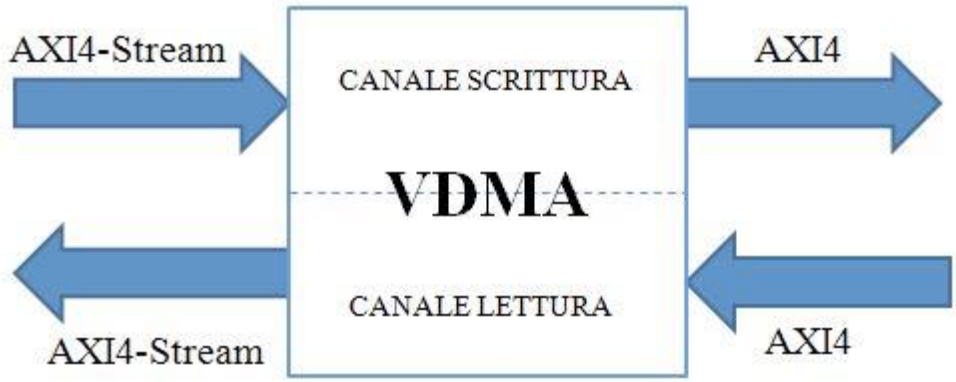


Figure 4-19: Interfacce VDMA

Per risparmiare risorse della PL è stato disattivato completamente il canale di lettura dei VDMA, poiché non utilizzato in questo progetto.

5. Linaro (Linux S.O.)

5.1.Preparazione SD

Per eseguire il boot di un altro sistema operativo, non ramdisk come quello preinstallato sulla scheda, è necessario preparare la scheda SD. Il primo passo è quello di formattarla adeguatamente:

1. Collegare la scheda SD al lettore
2. Installare un programma apposito per il partizionamento: noi abbiamo deciso di utilizzare **GParted** su Linux, per comodità di installazione e facilità di utilizzo.
3. Avviamo il programma di partizionamento
4. Selezionare la SD card (/**dev/sdb**) e cancellare tutte le partizioni
5. Creare una piccola partizione **FAT32** (o FAT16) che contenga i file necessari per il boot. Ricordarsi di lasciare 4MB liberi prima della partizione. Nominarla **BOOT** per comodità. La dimensione non è fondamentale, ma è consigliabile non farla oltre i 64MB che sono più che sufficienti.
6. Creare un'uteriore partizione **EXT4** (ma anche EXT3/EXT2) nel rimanente spazio non allocato della scheda SD e nominarla **rootfs**. Questa partizione conterrà il *FILE SYSTEM* linux.
7. Lanciare il partizionamento effettivo

Se tutto è andato come ci aspettiamo, dovremmo ritrovarci le due partizioni:

```
/media/<devicename>/BOOT
/media/<devicename>/rootfs
```

5.2.Installazione File System Linaro

Linaro è una distribuzione opensource di Linux, basata su Ubuntu, mirata a fornire una base stabile e testata per l'integrazione di un'architettura ARM.

Adesso non rimane che ottenere il tarball del filesystem Linaro della nostra distribuzione preferita. Possiamo scaricarla direttamente da http://releases.linaro.org/ cliccando sulla versione desiderata e poi cercando in Ubuntu/preciseimages. Possiamo trovare una versione con desktop grafico e che è stata testata sulla ZedBoard qui: *http://releases.linaro.org/12.09/ubuntu/preciseimages/ubuntudesktop*

Scompattiamo il contenuto dell'archivio scaricato:

```
sudo tar zxf linaropreciseubuntudesktop20120923436.tar.gz
```

Spostiamoci nella cartella binary/boot/filesystem.dir del contenuto scompattato

```
cd binary/boot/filesystem.dir
```

Dal momento che molti file nel filesystem di root hanno attributi speciali, è consigliato usare il comando rsync per copiarli sulla SD, in modo da avere la sicurezza di mantenere tali attributi per ogni file; questo processo impiegherà un po' di tempo a seconda della velocità della scheda SD.

```
sudo rsync -a --progress ./ /media/<devicename>/rootfs/
```

Per assicurarsi che tutti i file siano stati sincronizzati correttamente sulla scheda SD, è necessario smontare la partizione in modo sicuro /media/<devicename>/rootfs prima di scollegare la scheda SD. Potrebbero essere necessari diversi minuti per sincronizzare tutto correttamente.

```
sudo umount /media/<devicename>/rootfs/
```

5.3. Generazione dei file di boot

Per rendere possibile tutto ciò servono 3 file nella partizione di boot della scheda sd:

- **Boot.bin**: racchiude al suo interno FSBL, Uboot e bitstream di configurazione.
- **ZImage**: immagine compressa del kernel di linux
- **devicetree.dtb**: file che descrive le componenti hardware del sistema mediante una struttura ad albero, svolge il compito di descrittore dell'hardware che nei normali sistemi intel è svolto dal BIOS di sistema: mappare i dispositivi nell'area di indirizzamento del kernel, associarci driver se possibile e passare al kernel una serie di argomenti di boot.

E il File system precedentemente scompattato nell'altra partizione.

Al variare della configurazione hardware nel progetto *Vivado* è necessario ricreare il file boot.bin, visto che il bitstream sarà cambiato, e il devicetree; il resto dei file rimane invece intoccato.

Per generare i tre file è necessario l'SDK di Vivado e ,solo se optiamo per la compilazione manuale del kernel linux e del file uboot, la suite di tool di compilazione e sviluppo base per linux ottenibile digitando su terminale in un sistema operativo linux:

```
sudo apt-get install build-essential
```

Per comodità è possibile aggiungere in coda al proprio file ~/.bashrc le seguenti linee

```
export ARCH=arm
export CROSS_COMPILE=armxilinxlinuxgnueabiexport
PATH=/opt/Xilinx/SDK/2014.3.1/gnu/arm/lin/bin::$PATH
```

Che settano due variabili d'ambiente ed aggiungono al path i tools di crosscompilazioni rilasciati con l'SDK di Xilinx. Sono inoltre necessarie le repository contenenti il codice sorgente del kernel linux e dell'uboot scaricabili rispettivamente con i seguenti comandi su terminale.

```
git clone https://github.com/Xilinx/linuxxlnx.git
git clone https://github.com/Xilinx/ubootxlnx.git
```

Nel caso in cui git non sia installato nella macchina in uso è possibile provvedere con il comando

```
sudo apt-get install git
```

5.3.1. Generazione del file boot.bin

Il file *Boot.bin* può essere generato direttamente dall'sdk e racchiude al suo interno 3 componenti indipendenti: FSBL, Bitstream e Uboot che dovremmo creare uno alla volta:

- **FSBL:** può essere generato in maniera completamente automatizzata nell'*Sdk*
    1. Creare un nuovo progetto nell'*SDK* dal menu File>New>Application project
    2. Dare un nome al progetto, lasciare tutto il resto a default e cliccare su next
    3. Selezionare Zynq FSBL tra i teamplate disponibili e completare la creazione.

Se tutto è andato a buon fine all'interno del progetto nella cartella Debug sarà stato creato un file NomeDelProgetto.elf che è l'FSBL relativo al design sintetizzato.

● Bitstream: Il bitstream file è gia stato generato per noi da vivado e si trova nella cartella:

`NomeProgetto/NomeProgetto.sdk/NomeDesign_wrapper_hw_platform/nomeWrapper.bit`

● Uboot: Per la compilazione dell'uboot dobbiamo andare nella cartella ubootxlnx precedentemente scaricata e da terminale digitare:

```
make zynq_zed_config
make
```

Rispettivamente per configurare la repository per la creazione del file adatto alla zedboard e per la compilazione vera e propria. Al termine del processo in caso di succeso nella cartella sarà creato un file di nome uboot, l'unica cosa che resta da fare è aggiungergli l'estensione .elf rinominandolo in modo che sia visibile da Vivado. A questo punto abbiamo tutto ciò che ci serve per generare il file BOOT.bin possiamo quindi tornare sull'sdk e:

1. Cliccare su Xilinx Tools>Create Zynq Boot Image.
2. Selezionare Create New BIF file.
3. Nella casella Output BIF File Path inserire il path dove verrà salvato il file boot.bin.
4. Aggiungere mediante il pulsante add i tre file creati precedentemente in questo ordine: FSBL, Bitstream.bit e Uboot.elf.
5. Completare la creazione con "Create Image"

Al termine del processo nella cartella di output specificata verrà creato il file Boot.bin.

5.3.2. Generazione del file z-image

Il file z-image è ottenuto dalla compilazione del kernel linux, per crearlo:

1. Spostarsi nella cartella linux-xlnx scaricata precedentemente.
2. Eseguire uno a scelta tra:
   - `make xilinx zynq defconfig`→ configurazione standard del kernel.
   - `make menuconfig`→ apre un menu che permette di decidere manualmente quali moduli aggiungere o togliere al kernel.
3. Compilare il kernel con il comando:

   `make`
4. Completata la compilazione nella cartella linux-xlnx/arch/arm/boot saranno generati i file uImage e zImage necessari.

A questo punto l'unico file rimasto da creare è il *devicetree* a cui è dedicata una spiegazione più approfondita nel prossimo capitolo.

5.3.3. Generazione devicetree

Il file di Device Tree rappresenta una struttura dati per descrivere una piattaforma hardware. Il motivo per cui viene utilizzato è molto semplice: piuttosto che incorporare ogni dettaglio dei dispositivi hardware all'interno del sistema operativo, risulta di maggiore convenienza descrivere tali dettagli in un file opportuno che verrà passato al sistema operativo in fase di boot.

Appare evidente come ciò faciliti lo sviluppo e il testing di un progetto. La struttura del device tree è quella di un semplice albero di nodi con nome e di loro proprietà: ogni nodo può avere diverse proprietà e nodi figli. Le proprietà sono coppie chiavevalore e possono essere di qualsiasi tipo.

Chiaramente la struttura utilizzata per il device tree deve rispettare regole e convenzioni ben definite, in modo da poter essere "compresa" dal sistema operativo. In particolare sono stati predefiniti dei "bindings" per un gran numero

di dispositivi esistenti, ossia descrizioni di come tali dispositivi debbano essere rappresentati nel device tree.

Una volta esportato l'hardware da Vivado (includendo il bitstream), dopo aver lanciato l'SDK, è possibile lanciare la generazione del device tree creando un nuove BSP opportuno:

- SDK Menu:
  `File > New > Board Support Package > Board Support Package OS: devicetree > Finish`

A questo punto apparirà una finestra di BSP settings, attraverso la quale sarà possibile impostare alcune proprietà del device tree generato. Tale finestra di settings sarà reperibile anche selezionando il file system.mss del Device Tree BSP e cliccando "Modify this BSP's Settings".

Sono di interesse in particolare due proprietà:

- "bootargs", specifica gli argomenti passati al kernel in fase di boot
- "console device", specifica quale dispositivo di output seriale sarà utilizzato

Si rimanda ai paragrafi successivi per una spiegazione dei contenuti di tali proprietà. Ad esecuzione conclusa sarà possibile esaminare i file .dts e .dtsi (dts include) generati in automatico e situati in:

`<SDK workspace>/device_tree_bsp_0/ folder.`

Si noti che vi sono alcuni file .dts già generati e resi disponibili da Xilinx nella directory `Linux-xlnx/arch/<architecture>/boot/dts/.`

I file .dts e .dtsi sono semplici file di testo e necessitano di essere compilati in un unico file binario con estensione .dtb, che sarà poi utilizzato in fase di boot.

Per compilare e decompilare si utilizza il tool dtc, reso disponibile nel repository Linux-xlnx, in particolare nella directory `linux-xlnx/scripts/dtc`.

Lo stesso tool può essere ottenuto sia per compilare (da .dts a .dtb) che per decompilare (da .dtb a .dts):

- COMPILAZIONE: occorre specificare come input solo il file .dts, in quanto i file .dtsi sono inclusi in modo automatico. Eseguire il seguente comando:

```
dtc –I dts –O dtb –o <devicetree name>.dtb <devicetree name>.dts
```

- DECOMPILAZIONE: è sufficiente eseguire lo stesso comando, avendo cura di invertire gli argomenti di input e output:

```
dtc –I dtb –O dts –o <devicetree name>.dtb <devicetree name>.dts
```

Il file .dtb ottenuto in seguito alla fase di compilazione è pronto per essere copiato sulla partizione di boot della scheda SD.

La procedura di decompilazione può risultare utile per esaminare file .dtb già compilati, resi disponibili da Xilinx o da altri produttori.

6. Progetto Software

In questa parte del lavoro, il obiettivo è prendere questi immagini che sono memorizzati nella memoria della FPGA per la trasmissione anche a un computer via ethernet. Cosi, si possono vedere queasi “in tempo reale”. Para potere fare questo, essa ha ultilizzato una presa strumento chiamato socket.

6.1.Spazio d'Indirizzamento

Gli indirizzi sono resi disponibili al programmatore tramite il file system.xml, facilmente accessibile da SDK; gli indirizzi base sono raccolti anche, come costanti ed assieme ad altri parametri, nell’header file xparameters.h e, quindi, direttamente utilizzabili nel codice.

Durante la fase di assegnamento degli indirizzi si è dovuto tener conto dei vincoli previsti dallo Zynq: non tutti gli indirizzi, infatti, sono liberamente utilizzabili, come descritto in sezione 2.1. Nella tabella seguente è mostrato lo spazio d'indirizzamento di una delle soluzioni realizzate:

DISPOSITIVO	NOME COSTANTE	INDIRIZZO INIZIALE	INDIRIZZO FINALE	SPAZIO INDIRIZZAMENTO
DDR	DDR_RAM_BASEADDR	0x00100000	0x1FFFFFFF	512 MB
VDMA 0	AXI_VDMA_0_BASEADDR	0x43000000	0x4300FFFF	64 KB
UART	UART_0_BASEADDR	0xE0001000	0xE0001FFF	4 KB

Il numero di frame da bufferizzare rappresenta uno dei parametri del sistema e, per lo sviluppo del seguente progetto, si è preferito tenerlo contenuto. La bufferizzazione di due soli frame rappresenta un caso limite che, seppur possa essere più che sufficiente per la maggior parte delle applicazioni, si è preferito evitare, optando per la memorizzazione di tre frame.

Dovendo realizzare un sistema flessibile, in grado di supportare dinamicamente il cambiamento del tipo di sensori da collegare, gli indirizzi per i vari frame sono stati presi molto distanti tra loro.

Questi indirizzi sono mostrati nella tabella seguente:

VDMA	FRAME	INDIRIZZO INIZIALE	INDIRIZZO FINALE
0	0	0x08000000	0x08095FFF
	1	0x09000000	0x09095FFF
	2	0x0A000000	0x0A095FFF

6.2.Driver VDMA

Questi driver permettono la realizzazione di funzioni di più alto livello per la gestione del sistema, con particolare riferimento alla programmazione di più VDMA. Il comportamento del VDMA è completamente configurabile, ed è, quindi, possibile modificarne le specifiche a run-time. Oltre alla possibilità di scegliere la modalità di funzionamento desiderata, è possibile scegliere il numero di frame da bufferizzare ed impostare anche gli indirizzi della DDR utilizzati a tale scopo. Anche se questa opportunità non è stata sfruttata in questo lavoro di tesi, occorre notare che il canale di scrittura e quello di lettura di un VDMA non debbono per forza impiegare lo stesso buffer, quindi sono completamente indipendenti. Come questo sia possibile sarà mostrato in seguito, offrendo qualche dettaglio a riguardo.

I Driver di questo dispositivo sono contenuti nel BSP e suddivisi, come tipicamente avviene, in due file:

- xvdma.h: contiene la definizione delle strutture dati e delle funzioni
- xvdma.c: contiene l'implementazione delle funzioni

Le strutture dati principalmente utilizzate sono:

- XAxiVdma: rappresenta l'istanza del VDMA specifico, le informazioni principali in essa contenute riguardano: i canali attivi (lettura e scrittura), le relative strutture che ne contengono la configurazione e l'indirizzo iniziale del componente
- XAxiVdma_Config: raccoglie le informazioni riguardanti la configurazione hardware del componente; ogni VDMA deve avere la propria
- XAxiVdma_DmaSetup: contiene le informazioni necessarie alla configurazione software del componente per un singolo canale. Alcune di esse riguardano la tipologia del frame buffer, la dimensione dei frame del flusso video e gli indirizzi da utilizzare per la memorizzazione

La prima operazione da compiere consiste nell'inizializzazione del Driver. Successivamente si fanno partire i trasferimenti in DMA e, se fosse necessario, si impostano le routine di risposta agli interrupt. Quest'ultima possibilità non è stata sfruttata nello sviluppo di questo progetto e perciò non è illustrata.

Qui di seguito si mostra ora qual è la sequenza di operazioni da compiere per l'inizializzazione del driver. Occorre, innanzitutto, creare la struttura necessaria alla configurazione del VDMA; può essere fatto manualmente o attraverso la funzione:

XAxiVdma_LookupConfig (Device ID**);**

Il passo successivo consiste nella vera e propria inizializzazione del driver e del dispositivo, ciò è possibile attraverso la funzione:

XAxiVdma_CfgInitialize (XAxiVdma, XAxiVdma_Config, BaseAddr**)**

Per alcune possibili configurazioni del VDMA in questa fase occorrono ulteriori operazioni da compiere ma, non interessando direttamente il progetto, non sono illustrate e si può passare direttamente allo step successivo.

Per poter iniziare i trasferimenti in DMA rimangono da compiere alcune importanti sequenze d'istruzioni:

- **XAxiVdma_DmaConfig (**XAxiVdma, XAxiVdma_DmaSetup**)**: serve alla configurazione del singolo canale del VDMA; le relative informazioni di configurazione sono raccolte nella struttura dati XAxiVdma_DmaSetup. Ne esiste una per il canale di scrittura e una per il canale di lettura e, proprio grazie a queste, essi sono completamente indipendenti
- **XAxiVdma_DmaSetBufferAddr (**XAxiVdma, ChannelDir, Buffer_Addresses**)**: permette di impostare a piacimento gli indirizzi fisici utilizzati per la bufferizzazione dei frame; la struttura dati Buffer_Addresses conterrà, quindi, un numero di elementi pari al numero di frame gestiti
- **XAxiVdma_DmaStart (**XAxiVdma, ChannelDir**)**: con questa, terminata la configurazione, si possono iniziare i trasferimenti.

Il driver fornisce anche il supporto necessario alla fase di debug; innanzitutto, tutte le funzioni precedentemente descritte, restituiscono un esito che permette di sapere se l'operazione effettuata è andata a buon fine.

Un'altro utile strumento è rappresentato dalla funzione sotto riportata:

XAxiVdma_DumpRegister (XAxiVdma, ChannelDir**)**

Essa stampa i registri di stato e di controllo del VDMA. Una dei limiti principali del driver è rappresentato dall'assenza di meccanismi per la gestione della coerenza delle cache dati che deve essere, quindi, sotto la responsabilità dell'applicazione.

Finalmente dire che l’uso del driver su Linux e uguale che queste che abbiamo spiegato, ma riservando lo spazio di memoria per i buffer tramite il kernel Linux.

6.3.SERVER SOCKET IMAGE

Un socket è un metodo per la comunicazione tra un programa client e un programa server in una rete. Un socket è definito come il punto finale in una connessione. I socket vengono creati e ultilizzati con una serie di richieste o chiamate di funzione a volte chiamato la progrmamazione di applicazione prese di interfaccia API (Application Programming Interface).

Figure 6-1:Client cercando la conesione con il server in un socket

Figure 6-2: Server accettando la conesione con il cliente in un socket

In questo caso il socket (sia il client o il server) è programmato in codice C. È il server il responsabile di inviare le immagini, che si trovano nella memoria della FPGA, al cliente (in questo caso un PC).

I socket hanno sempre una serie di funzioni speciali che sono sempre uguale.

Cliente	Servidor
	socket()
	bind()
	listen()
	accept()
socket()	
[bind()]	
connect()	
send()	recv()
recv()	send()

Figure 6-3:Struttura di un socket orientato alla conessione.

6.3.1. Funzione Socket:

```
int socket (int domain, int type, int protocol);
```

Domain: Sarà impostato come AF_INET (ARPA utilizzare protocolli Internet), o AF_UNIX (se si desidera creare socket per il sistema di comunicazione interna) . Questi sono i più utilizzati, ma non unico

Type: Qui è necessario specificare il tipo di presa che si desidera utilizzare (Flow o Datagram). Le variabili sono elencati sono SOCK_STREAM o SOCK_DGRAM come vogliamo utilizzare i socket o Datagram flusso, rispettivamente.

Protocol: Qui , si può semplicemente impostare il protocollo a 0.

6.3.2. Funzione Bind:

```
int bind (int fd, struct sockaddr *my addr, int addrlen);
```

FD: È il descrittore di file presa restituito dalla chiamata a socket.

My_addr: È un puntatore a una struttura sockaddr

Addrlen: Contiene la lunghezza della struttura sockaddr a cui punta il puntatore my_addr. Deve essere impostato a sizeof (struct sockaddr)

La chiamata bind viene utilizzato quando le porte locali della nostra macchina sono nei nostri piani (di solito quando si utilizza la chiamata listen). La sua funzione essenziale è quella di associare un socket con una porta (in nostra macchina). Allo stesso socket restituirà -1 in caso di errore .

D'altra parte siamo in grado di rendere il nostro indirizzo IP e la porta vengono selezionati automaticamente

server.sin port = 0;
server.sin adder.s addr = INADDR_ANY

6.3.3. Funzione Connect:

```
int connect (int fd, struct sockaddr *serv addr, int addrlen);
```

FD: Deve essere impostato per il descrittore di file del socket , che è stato restituito dalla chiamata a socket.

Serv_addr: È un puntatore alla struttura sockaddr che contiene l'indirizzo IP di destinazione e la porta.

Addrlen: Analogamente a quanto accaduto con la funzione bind, questo argomento dovrebbe essere impostato a sizeof (struct sockaddr).

6.3.4. Funzione Listen:

```
int listen (int fd, int backlog);
```

FD: E ' il descrittore di file socket, che è stato restituito dalla chiamata a socket.

Backlog: Il numero di connessioni consentite.

6.3.5. Funzione Accept:

```
int accept (int fd, void *addr, int *addrlen);
```

FD: E ' il descrittore di file socket, che è stato restituito dalla chiamata a socket.

Addr: E ' un puntatore ad una struttura sockaddr_in nella quale in grado di determinare quale nodo noi e da quale porta sta contattando.

Addrlen: Analogamente a quanto accaduto con la funzione bind, questo argomento dovrebbe essere impostato a sizeof (struct sockaddr).

6.3.6. Funzione Send:

```
int send(int fd, const void *msg, int len, int flags);
```

FD: E ' il descrittore di file presa con il quale si desidera inviare i dati.

MSG: Si tratta di un puntatore che indica i dati che si desidera inviare.

Len: È la lunghezza dei dati che si desidera inviare (in byte).

Flags: Deve essere impostato su 0 [8].

6.3.7. Funzione Recv:

```
int recv(int fd, void *buf, int len, unsigned int flag);
```

FD: È il descrittore del socket con cui i dati vengono letti .

Buf. È il buffer per ricevere le informazioni viene salvato .

Len: È la lunghezza massima che può avere il buffer .

Flag: A questo punto, dovrebbe essere impostato come 0 .

6.3.8. Funzione close:

```
close (fd);
```

La funzione close è utilizzato per chiudere la connessione.

Codice server:

```
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>

#define PORT 1990

int parseARGS(char **args, char *line)
{
    int tmp = 0;
    args[tmp] = strtok(line, ":" );
    while ((args[++tmp] = strtok(NULL, ":" )) != NULL);
    return tmp -1;
}

int main()
{
    char *header[4096];
    int listenSOCKET, connectSOCKET;
    socklen_t clientADDRESSLENGTH;
    struct sockaddr_in clientADDRESS, serverADDRESS;
    char recvBUFF[4096];
    char *filename, *filesize;
    FILE *recvFILE;
    int received = 0;
    char tempstr[4096];

    int percent;

    listenSOCKET = socket(AF_INET, SOCK_STREAM, 0);
    if (listenSOCKET <0)
    {
        printf("Cannot create socket\n");
        close(listenSOCKET);
        return 1;
    }

    serverADDRESS.sin_family = AF_INET;
    serverADDRESS.sin_addr.s_addr = htonl(INADDR_ANY);
    serverADDRESS.sin_port = htons(PORT);

    if (bind(listenSOCKET, (struct sockaddr *)&serverADDRESS,
sizeof(serverADDRESS))<0)
    {
        printf("Cannot bind socket\n");
        close(listenSOCKET);
        return 1;
    }

    listen(listenSOCKET, 5);
    clientADDRESSLENGTH = sizeof(clientADDRESS);
    connectSOCKET = accept(listenSOCKET,(struct sockaddr
*)&clientADDRESS, &clientADDRESSLENGTH);
    if (connectSOCKET<0)
    {
        printf("Cannot accpet connection\n");
        close(listenSOCKET);
        return 1;
    }
```

```
    while(1)
    {
            printf("Conectado!!");
        if(recv(connectSOCKET, recvBUFF, sizeof(recvBUFF),0))
        {
            if(!strncmp(recvBUFF,"FBEGIN",6))
            {
            recvBUFF[strlen(recvBUFF)-2] = 0;
            parseARGS(header, recvBUFF);
            filename = header[1];
            filesize = header[2];
            }
        recvBUFF[0] = 0;
        recvFILE = fopen(filename, "w");
            printf("Dim imagen: %s\n", filesize);
        percent = atoi(filesize)/100;
        printf("Current percent value: %d\n",percent);
        while(1)
        {
            if(recv(connectSOCKET,recvBUFF,1,0) != 0)
            {
                fwrite(recvBUFF,sizeof(recvBUFF[0]),1, recvFILE);
                recvBUFF[0] = 0;
            }
            else
            {
                close(listenSOCKET);
                return 0;
            }
        }
        close(listenSOCKET);
        }
    else
    {
        printf("Client dropped connection\n");
    }
    return 0;
    }
}
```

Codice Cliente:

```
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>

int fileSEND(int PORT, char *lfile, char *rfile)
{
    int socketDESC;
    struct sockaddr_in serverADDRESS;
    FILE *file_to_send;
    int ch;
    char toSEND[1];
    char remoteFILE[4096];
    int percent;



    socketDESC = socket(AF_INET, SOCK_STREAM, 0);
    if(socketDESC<0)
    {
        printf("Cannot create socket\n");
        return 1;
    }

    serverADDRESS.sin_family = AF_INET;
    serverADDRESS.sin_addr.s_addr = inet_addr ("192.168.1.78");
    serverADDRESS.sin_port = htons(PORT);

    if(connect(socketDESC,(struct sockaddr
*)&serverADDRESS,sizeof(serverADDRESS))<0)
    {
        printf("Cannot connect\n");
        return 1;
    }

    file_to_send = fopen(lfile, "r");
    if(!file_to_send)
    {
        printf("Error opening file\n");
        close(socketDESC);
        return 0;
    }
    else
    {
        long fileSIZE;
        fseek(file_to_send,0,SEEK_END);
        fileSIZE=ftell(file_to_send);
        rewind(file_to_send);

      printf("El tamano de la imagen es: %ld\n",fileSIZE);

        sprintf(remoteFILE,"FBEGIN:%s:%ld\r\n",rfile,fileSIZE);
        send(socketDESC,remoteFILE,strlen(remoteFILE),0);
        percent = fileSIZE/100;
        while((ch=getc(file_to_send))!=EOF)
        {
            toSEND[0]=ch;
            send(socketDESC,toSEND,1,0);
```

```
            }
        }
        fclose(file_to_send);
        close(socketDESC);
        return 0;
}

int main()
{
        fileSEND(1990, "name.jpeg", "exito.jpeg");
        return 0;
}
```

7. Conclusioni

Dopo il nostro lavoro, abbiamo scoperto che questa scheda FPGA di circuiti riconfigurabili della famiglia Zynq-7000 All Programmable SoC è un instrumento con molto potenziale che offre molte possibilità di creazione di aplicazioni in diversi ambiti.

Abbiamo imparato pure la dificoltà sulla gestione e creazione di un progetto di questo genere e anche come l'uso di queste tipo di FPGA rende più semplice lo svilupo facendo una riduzione del costo e del tempo.

Facendo uso del VDMA è stato possibile gestire e controlare il flusso di frame generati per il TPG fino alla DDR3, realizzando così un frame buffer circolare. Dobbiamo dire che la esistenza dei driver (sia per Linux che per *baremetal*) rende molto più semplice questo lavoro e fa che il VDMA sia più efficiente.

Nel caso di Linux, la scelta di Linaro è stata perche questa distribuzione è molto leggera per la nostra FPGA e i loro ARM-Core e anche perche offre risorse abbastanza per la inclusione del driver e le *library* per la estrazione dei frame e il loro invio tramite *ethernet* facendo la FPGA ancora più efficiente.

Finalmente, dobbiamo dire che il nostro progetto non è un lavoro rigido nel senso che offre flessibilità per futuri sviluppatori e per aggiungere modifiche sia di resoluzione della telecamera, sia della quantità di telecamere oppure del proprio server. Per questo, il nostro progetto finisce qui, ma la sua funzionalità potrebbe essere sfruttata nel futuro.

8. Bibliografia

- *http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html*
- *Xilinx, ds190-Zynq-7000-Overview.pdf*
- *ARM, Bus Axi specs.pdf*
- *www.zedboard.org*
- *https://www.linaro.org/*
- *http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf*
- *http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf*
- *http://www.xilinx.com/support/documentation/application_notes/xapp742-axi-vdma-reference-design.pdf*
- *https://www.digilentinc.com/Data/Products/EMBEDDED-LINUX/ZedBoard_GSwEL_Guide.pdf*
- *Xilinx, axi_reference_guide.pdf*
- *Xilinx, ds768_axi_interconnect.pdf*