Pynq Motion Detective

Progetto Sistemi Digitali M

Link repository Github

Febbraio 2024

Luca Cimino
Federico Mingarelli
Stefano Spadari

Indice

Capitolo 1

Introduzione

Il progetto ha come obiettivo la realizzazione di un sistema di sorveglianza basato sulla board PYNQ-Z2. Il sistema usa la motion detection per rilevare i movimenti nella scena monitorata, e quando rileva un movimento invia un flusso video cifrato un host remoto tramite una connessione di rete. Le operazioni crittografiche sono gestite da un IP Core di cifratura implementato sull'FPGA della PYNQ. Ogni frame del flusso video viene cifrato prima di essere trasmesso sulla rete, garantendo la sicurezza e la privacy dei dati trasmessi.

Il progetto prevede anche un client remoto che riceve il flusso video e dopo averlo decifrato lo mostra all'utente.

Nei prossimi capitoli vengono approfonditi i dettagli relativi allo sviluppo dell'applicazione, trattando il rilevamento del movimento, la crittografia, e infine i risultati ottenuti.

1.1 Pynq

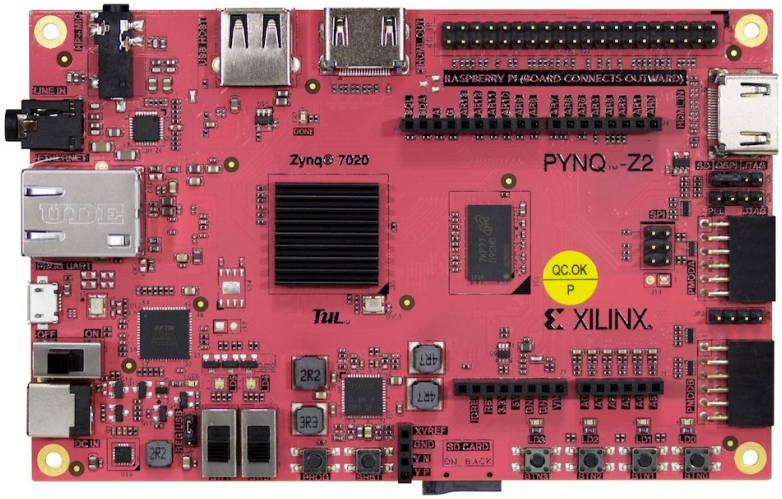

Figura 1.1: PYNQ

PYNQ è un progetto open source sviluppato da AMD, progettato per semplificare lo sviluppo di applicazioni ad alte prestazioni che richiedono un elevato numero di calcoli. Dispone di uno Zynq-7000 SoC, che grazie alla combinazione di un processore ARM Cortex-A9 dual-core e di una FPGA programmabile, offre una piattaforma versatile per la creazione di sistemi embedded avanzati.

Molteplici applicazioni possono beneficiare dell'utilizzo di PYNQ e delle sue capacità avanzate di elaborazione hardware e software, tra cui:

- **Accelerazione hardware di algoritmi:** sfruttare la capacità dell'FPGA di eseguire operazioni in parallelo per accelerare l'esecuzione di algoritmi complessi, migliorando le prestazioni complessive dell'applicazione.
- **Elaborazione video:** implementazione di algoritmi per l'elaborazione di video ad alta risoluzione e velocità di frame elevate.
- **Elaborazione del segnale in tempo reale:** Realizzazione di sistemi di elaborazione del segnale in grado di operare in tempo reale, con tempi di risposta rapidi e prestazioni elevate.

La principale caratteristica del progetto PYNQ è quella di proporre l'utilizzo del linguaggio Python, che permette agli sviluppatori di scrivere codice ad alto livello, rendendo l'intero processo di sviluppo più accessibile anche a coloro che non hanno familiarità con il design hardware.

Per integrarsi con la tecnologia FPGA, viene messo a disposizione il modulo Overlay, che astrae i componenti hardware, gli IP core, per poter invocare le funzioni accelerate su FPGA in maniera del tutto trasparente.

1.2 Computer Vision e Motion Detection

La computer vision è un campo dell'informatica e dell'ingegneria che si occupa dello sviluppo di sistemi in grado di acquisire, elaborare, analizzare e comprendere immagini e video digitali. L'obiettivo della computer vision è quello di dotare i computer della capacità di "vedere" e interpretare il mondo visivo circostante in modo simile agli esseri umani.

Questa disciplina si basa su una vasta gamma di tecnologie e algoritmi, tra cui image processing, pattern recognition, image segmentation, motion tracking, 3D reconstruction e infine motion detection. Quest'ultima si basa sull'identificazione di cambiamenti significativi nei frame di un flusso video, consentendo di individuare la presenza di oggetti in movimento all'interno della scena monitorata. Questa tecnica trova ampio impiego in una vasta gamma di applicazioni, tra cui domestic security, monitoraggio industriale, sorveglianza pubblica e analisi del traffico.

Nel contesto dell'applicazione in esame, la motion detection riveste un ruolo cruciale. L'obiettivo è quello di implementare un sistema in grado di analizzare in tempo reale un flusso video proveniente da una telecamera di sorveglianza e rilevare eventuali movimenti nella scena monitorata.

Questo processo consente di identificare e segnalare tempestivamente situazioni di interesse, come intrusioni non autorizzate o attività sospette, fornendo agli utenti un mezzo efficace per la sicurezza e il monitoraggio.

1.3 Cifratura dei dati

La trasmissione di video da una telecamera di sorveglianza rappresenta un potenziale rischio, in quanto un attaccante che intercetta questo flusso potrebbe ottenere accesso a filmati privati e sensibili, mettendo a repentaglio la sicurezza e la privacy del luogo monitorato.

Risulta quindi fondamentale adottare misure di sicurezza adeguate, come la cifratura del flusso video, per proteggere la riservatezza delle immagini e permetterne la visione solamente a chi autorizzato.

In questo progetto, per la cifratura dei frame, è stato scelto AES (*Advanced Encryption Standard*), un algoritmo crittografico a chiave simmetrica selezionato dal

NIST ed adottato come standard dal governo degli Stati Uniti. L'algoritmo AES opera su blocchi di testo in chiaro da 128 bit, applicando un serie di operazioni di sostituzione e trasposizione in maniera iterata. La chiave simmetrica usata può essere di 128, 192 o 256 bit in base al livello di sicurezza che si desidera.

Tra i grandi vantaggi di AES si evidenziano l'elevato grado di sicurezza a fronte di ogni attacco conosciuto, le ottime prestazioni sia se sviluppato in software sia in hardware, e la bassa richiesta di memoria che ne permette l'implementazione anche su dispositivi con basse risorse.

Capitolo 2

Sviluppo

2.1 Infrastruttura e Motion Detection

L'applicazione è un sistema distribuito, con un server che si presuppone essere sempre attivo, posto come un demone sulla PYNQ. Il compito del server è monitorare la scena ripresa dalla telecamera, e quando rileva un movimento, spedire il flusso video cifrato al client connesso in ascolto.

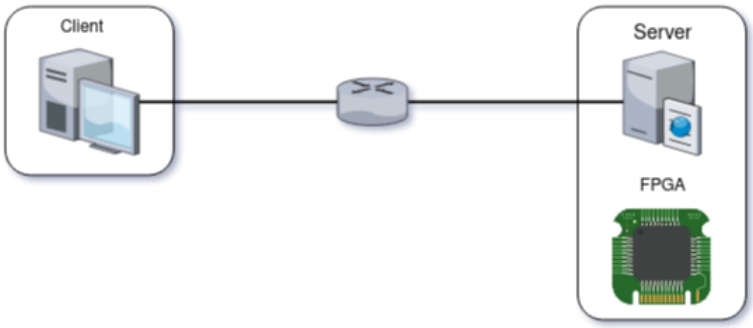

Figura 2.1: Client-Server Architecture

Andando più nel dettaglio, abbiamo scelto il protocollo di trasporto UDP, dove ogni frame è inserito in un singolo pacchetto. Pur accettando il rischio di perdita di dati, questo approccio è finalizzato alla creazione di uno stream continuo per tutta la durata necessaria.

Precedentemente all'invio del frame, avviene la fase di analisi, ovvero la verifica di un possibile movimento. Ogni frame viene catturato dalla fotocamera e viene confrontato con il precedente.

L'algoritmo per rilevare un movimento si basa sull'utilizzo della differenza tra i 2 frame, e una serie di operazioni di ottimizzazione del calcolo.

Per questa parte si è optato per l'utilizzo della libreria Python *openCV*.

Attraverso le funzioni offerte da *openCV*, per ogni frame rilevato dalla fotocamera, possiamo andare a leggerlo e ad assegnarlo ad una variabile.

In seguito effettuiamo una conversione da BGR a greyScale, per poi passare i due frame alla funzione *get_mask()*.

```
def get_mask(frame1, frame2, kernel=np.array((9, 9), dtype=np.uint8)):
    frame_diff = cv2.subtract(frame2, frame1)

    # blur the frame difference
    frame_diff = cv2.medianBlur(frame_diff, 3)

    mask = cv2.adaptiveThreshold(frame_diff, 255,
    ↪  cv2.ADAPTIVE_THRESH_GAUSSIAN_C, \
                                 cv2.THRESH_BINARY_INV, 11, 3)

    mask = cv2.medianBlur(mask, 3)

    # morphological operations
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel, iterations=1)

    num_diff_pixels = np.sum(mask == 255)

    return mask, num_diff_pixels
```

Da come si può vedere, la funzione *get_mask()* si occupa di ottenere la maschera binaria che identifica i pixel in movimento tra i due frame e il numero di pixel identificati come diversi.

La prima operazione calcola la differenza tra frame2 e frame1 utilizzando la sottrazione di immagini *cv2.subtract()*.

cv2.medianBlur() applica un filtro (kernel 3x3) che restituisce per ogni finestra 3x3 il valore mediano alla differenza dei frame per ridurre il rumore.

cv2.adaptiveThreshold() applica una soglia alla differenza tra frame, ottenendo una maschera binaria che evidenzia i pixel corrispondenti ai cambiamenti significativi.

cv2.morphologyEx() chiude eventuali lacune all'interno degli oggetti identificati come in movimento nell'immagine binaria.

Infine viene calcolato il numero di pixel che differiscono.

Quando viene rilevata una sequenza di frame diversi in un intervallo ristretto, si deve attivare la trasmissione tra server e client per l'invio dello stream video.

Finchè il sistema non rileva che i movimenti sono terminati, lo stream deve rimanere aperto e ogni frame deve essere spedito da server a client.

Per effettuare l'invio dei frame tra un nodo e l'altro attraverso protocollo UDP ci siamo dovuti assicurare di rispettare i vincoli introdotti dalla dimensione massima dei pacchetti. In questo senso, si è deciso di effettuare prima un ridimensionamento e poi una conversione di colori per far si che ad ogni pacchetto corrisponda un frame.

```
# resize (640,480) -> (240,180)
frame = cv2.resize(frame, (COMPRESSED_WIDTH, COMPRESSED_HEIGHT))

# RGB -> YUV
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV_I420)

# Ottieni i dati del frame come bytes
frame_data = frame.tobytes()

```

Come si può vedere da queste righe di codice, ci siamo avvalsi nuovamente delle funzioni offerte da *openCV*. Per l'invio delle immagini abbiamo voluto trovare una combinazione che ci consentisse di perdere risoluzione il meno possibile, ma che ci permetesse di restare dentro i limiti. Per questo motivo abbiamo scelto di effettuare un ridimensionamento da 640x480 a 240x180, abbinato alla conversione da BGR in YUV_420.

YUV è un sistema di codifica a colori ampiamente utilizzata nell'ambito della compressione video e della trasmissione di immagini, specialmente in contesti in cui è necessario ridurre lo spazio di archiviazione o la larghezza di banda richiesta per l'invio dei dati video. La codifica digitale di video e immagini Y'UV definisce uno spazio colore utilizzando un componente di luminanza (Y') e due componenti di crominanza (UV). Per convertire un'immagine dal formato RGB al formato YUV, il primo passo consiste nella conversione dell'immagine in scala di grigi per ottenere il canale Y, che rappresenta la luminanza. Successivamente, le componenti di crominanza U e V vengono calcolate mediante una tecnica basata sulla sottrazione della luminanza dalle componenti RGB. Infine, i canali di colore U e V possono essere sottocampionati per ridurre la dimensione dei dati. In particolare esistono diverse versione di YUV:

- ***4:4:4*** *: In questa configurazione, tutte le componenti (Y', U, V) sono campionate alla stessa risoluzione, senza alcuna sottocampionatura.*

- ***4:4:0*** *: In questa configurazione, la componente di luminanza (Y') viene mantenuta alla massima risoluzione, mentre le componenti di crominanza (U, V) vengono dimezzate lungo l'asse orizzontale, riducendo la quantità di dati.*
- ***4:2:0*** *: Questa configurazione dimezza ulteriormente le componenti di crominanza (U, V) rispetto alla configurazione 4:4:0, riducendo ulteriormente le dimensioni dei dati.*

Nel nostro caso si è scelto di utilizzare la 4:2:0 perché permette di perdere maggiori informazioni sul colore, la cui differenza è meno impercettibile e meno importante, in modo tale da poter mantenere una risoluzione più alta. Questa codifica utilizza solamente 12 bit per ogni pixel dell'immagine, contro i 24 bit della codifica RGB, riducendo la dimensione complessiva di un frame del 50%.

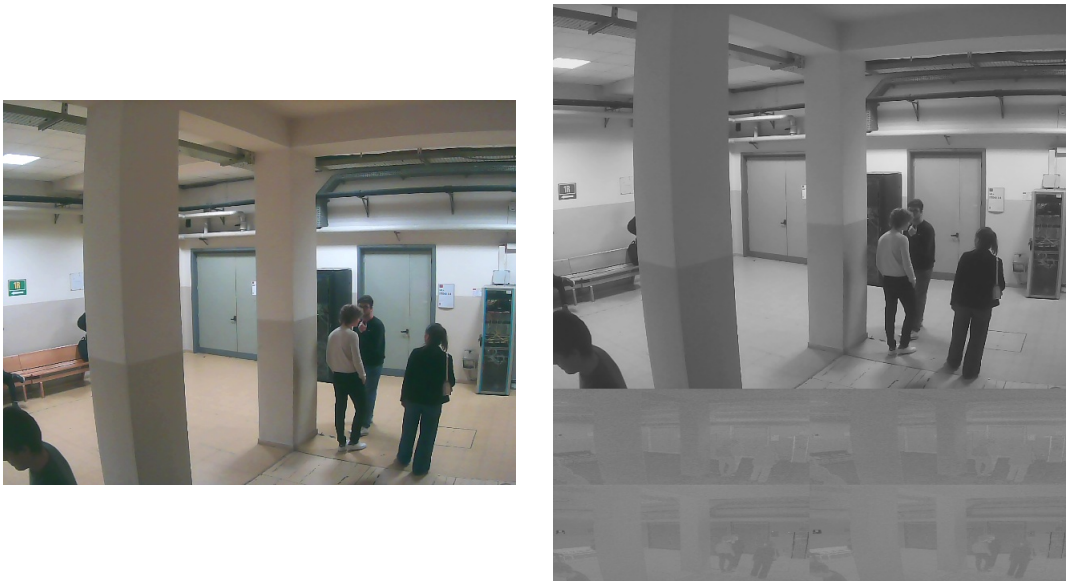

Figura 2.2: Confronto immagine RGB e immagine YUV 4:2:0

Dopo la conversione a YUV, il frame viene tradotto in byte e dopo essere stato cifrato, viene spedito. Il client resta in ascolto e riceve i frame, e dopo averli decifrati, fa la conversione inversa, quindi da YUV_420 a BGR, ridimensiona l'immagine e la mostra a schermo.

```
1 frame_rgb = cv2.cvtColor(frame, cv2.COLOR_YUV2BGR_I420)
2
3 frame_rgb = cv2.resize(frame_rgb, (640, 480))
4
5 cv2.imshow(window_name, frame_rgb)
```

Al termine dello sviluppo dell'applicazione, dopo un'attenta analisi si è deciso di fare una modifica alle dimensioni del frame su cui viene calcolata la maschera binaria.

Fino ad ora tutte le operazioni eseguite nella *get_mask()* sono state fatte utilizzando un frame di dimensione 640x480, questo comporta ad un tempo di esecuzione medio di quell'immagine di 0.1929 s. Ridimensionando la finestra a 240x180 pixel, l'elaborazione complessiva richiede in media solo 0.04282 s.

2.2 Cifratura di immagini su FPGA

2.2.1 Modalità di cifratura ECB

Le immagini sottoposte a cifratura sono in formato YUV 4:2:0 con dimensioni 240x180 pixel, per un totale di 64800 bytes. Siccome l'algoritmo AES opera su blocchi in input di 128 bit, occorre suddividere il testo da cifrare in blocchi di questa dimensione prima di poter procedere con la cifratura. Per cifrare dati di grandi dimensioni usando AES esistono diverse modalità; la più semplice ed anche quella scelta per l'implementazione in questo progetto è la modalità ECB (*Electronic Code Book*). Tale modalità prevede di cifrare ogni blocco di 128 bit in modo indipendente dagli altri usando la stessa chiave simmetrica.

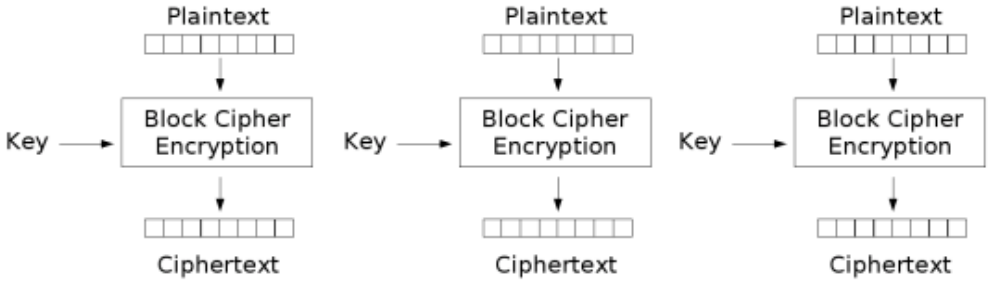

Figura 2.3: Modalità di cifratura ECB

Questa modalità di cifratura ha importanti vantaggi:

- permette la cifratura in parallelo dei singoli blocchi;
- in seguito alla modifica di un blocco cifrato in trasmissione, l'errore non viene propagato anche sugli altri blocchi;

ma data la sua semplicità, ha anche una pericolosa vulnerabilità: blocchi identici di testo in chiaro producono blocchi identici di testo cifrato. Tuttavia, per il nostro progetto, questo aspetto non spaventa particolarmente in quanto nelle immagini reali, luci e ombre rendono molto rara la presenza di un elevato numero di pixel esattamente identici. Tale situazione, al contrario, è più facile che si verifichi con immagini disegnate digitalmente oppure durante la cifratura di documenti, nei quali la presenza ricorrente di una stessa frase o parola potrebbe rivelare informazioni segrete ad un possibile attaccante che intercetta il flusso di dati.

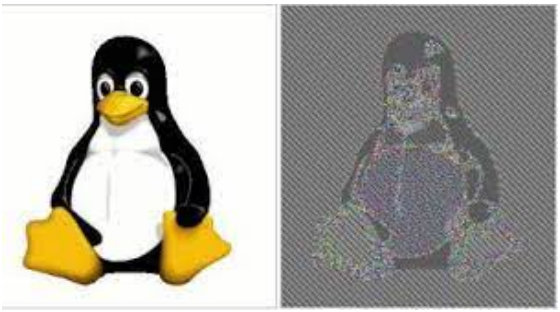

Figura 2.4: Vulnerabilità di ECB

Figura 2.5: Modalità ECB su immagini reali

2.2.2 High Level Synthesis

La sintesi ad alto livello permette di sviluppare un design hardware per FPGA a partire da una descrizione del comportamento espressa in linguaggio C, affiancata da direttive per il compilatore. Questo processo consente di tradurre efficacemente le funzionalità specificate nel codice C in una rappresentazione hardware descritta in linguaggio Verilog o VHDL. In questo modo, partendo da una descrizione più astratta del sistema, si semplificano il processo di sviluppo e si accelerando i tempi di progettazione.

La progettazione del modulo incaricato di eseguire la cifratura in modalità ECB delle immagini da trasmettere è stata svolta usando il software Vivado HLS. Questo strumento si occupa di eseguire la sintesi partendo da una funzione principale, detta *top function*. Ogni altra funzione invocata a partire dalla top function viene inclusa automaticamente nel processo di sintesi.

La dichiarazione della top function è presente nel file *main.h*, che definisce l'interfaccia del modulo e il tipo di dato *stream_type* usato negli stream di input e di output.

```
#include <hls_stream.h>
#include <stdint.h>
#include <ap_int.h>

#define BLOCK_SIZE 16 // bytes

struct stream_type {
        uint8_t data;
        ap_uint<1> user;  // USER signal
        ap_uint<1> last;  // TLAST signal
};

void AES_encryption(hls::stream<stream_type> &i_plaintext,
↪  hls::stream<stream_type> &cipher);

```

Il sorgente *main.cpp* contiene la definizione della top function. I due stream, argomenti della funzione, rappresentano rispettivamente l'input del modulo, che corrisponde al flusso di pixel da cifrare, e l'output del modulo, ovvero i pixel cifrati.

```
void AES_encryption(hls::stream<stream_type> &i_plaintext,
↪  hls::stream<stream_type> &cipher)
{
        #pragma HLS INTERFACE axis port=i_plaintext
        #pragma HLS INTERFACE axis port=cipher
        #pragma HLS INTERFACE ap_ctrl_none port=return
```

Entrambi gli argomenti sono definiti come interfacce AXI4-Stream. AXI4-Stream è un protocollo di comunicazione unidirezionale tra un dispositivo master, che invia i dati, e un dispositivo slave che li riceve. A seconda di come vengono utilizzate nel codice, Vivado HLS è in grado di capire da solo se una porta è master o slave, in particolare, nel nostro progetto la porta *i_plaintext* è slave, mentre la porta *cipher* è master. Tale protocollo non fa utilizzo degli indirizzi e si presta bene per situazioni in cui i dati vengono comunicati ed elaborati in un ordine ben preciso e sempre regolare, come proprio la cifratura a blocchi di un'immagine.

In un AXI4-Stream la comunicazione tra master e slave è regolata da un handshake, e per ogni ciclo di clock vengono trasferiti TDATA bits. Ogni trasferimento inizia con l'invio del segnale TVALID da parte del master. Lo slave risponde con il segnale TREADY, e prosegue a leggere tutti i dati finchè non incontra il segnale TLAST inviato dal master.

Mentre i segnali TVALID e TREADY sono sintetizzati e gestiti in automatico da Vivado HLS, il segnale TLAST non viene incluso nell'interfaccia se non esplicitamente usato. Per questo motivo il tipo di dato *stream_type* definito in

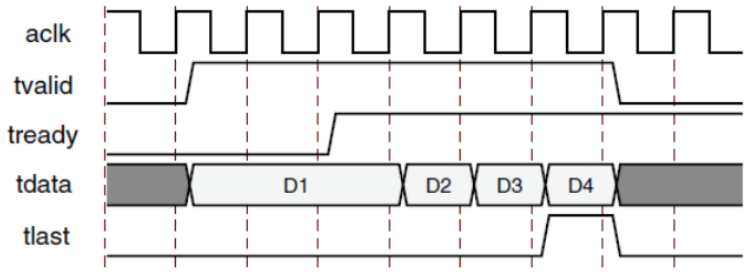

Figura 2.6: Comunicazione su canale AXI4-Stream

main.h prevede un campo *last* che deve essere esplicitamente comandato per ogni comunicazione sulla porta *cipher*.

Nella figura 2.7 si mostrano le interfacce del modulo prodotte della sintesi. Si noti come la direttiva *ap_ctrl_none* elimini i segnali di controllo per l'I/O, superflui avendo le interfacce AXI4-Stream.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	AES_encryption	return value
ap_rst_n	in	1	ap_ctrl_none	AES_encryption	return value
i_plaintext_TDATA	in	8	axis	i_plaintext_V_data	pointer
i_plaintext_TVALID	in	1	axis	i_plaintext_V_data	pointer
i_plaintext_TREADY	out	1	axis	i_plaintext_V_last_V	pointer
i_plaintext_TLAST	in	1	axis	i_plaintext_V_last_V	pointer
i_plaintext_TUSER	in	1	axis	i_plaintext_V_user_V	pointer
cipher_TDATA	out	8	axis	cipher_V_data	pointer
cipher_TVALID	out	1	axis	cipher_V_last_V	pointer
cipher_TREADY	in	1	axis	cipher_V_last_V	pointer
cipher_TLAST	out	1	axis	cipher_V_last_V	pointer
cipher_TUSER	out	1	axis	cipher_V_user_V	pointer

Figura 2.7: Interfacce IP Core di cifratura

Nell'esecuzione della top function, i dati letti dallo stream di input vengono suddivisi in blocchi di dimensione fissata e sottoposti alla cifratura con una chiave simmetrica di 128 bit cablata all'interno del modulo. Sfruttando il vantaggio della modalità di cifratura ECB di poter essere parallelizzata, il modulo viene progettato per poter eseguire la cifratura in parallelo di più blocchi di testo in chiaro. Attraverso la direttiva **HLS unroll** il compilatore di Vivado HLS viene istruito di creare multiple copie dell'hardware dedicato ad eseguire un'iterazione del loop al

fine di poter eseguire più iterazioni in parallelo. Un loop può essere *fully unrolled* o *partially unrolled*. Non specificando un parametro *factor* nella direttiva, il loop viene considerato *fully unrolled*, e questo porta alla creazione di una copia del corpo del loop per ogni iterazione, di modo che l'intero loop possa essere eseguito in parallelo.

```
1  loop: for(block = 0; block < 450; block++)
2  {
3  #pragma HLS unroll
4      // Lettura del plaintext dallo stream di input
5      for(i = 0; i < BLOCK_SIZE; i++) {
6          tmp = i_plaintext.read();
7          plaintext[i] = tmp.data;
8      }
9
10     // Cifratura
11     aes_cipher(plaintext, out, w);
12
13     // Scrittura sull'interfaccia AXI-Stream di output
14     for(i = 0; i < BLOCK_SIZE; i++) {
15         tmp.user = 1;
16         tmp.data = out[i];
17
18         if(i == BLOCK_SIZE-1 && block == 449)
19             tmp.last = 1;
20         else
21             tmp.last = 0;
22
23         cipher.write(tmp);
24     }
25 }
```

Nel nostro progetto viene effettuata la cifratura di 450 blocchi in parallelo, che corrispondono ad un totale di 7200 bytes. Questa scelta è dovuta al fatto che l'immagine da cifrare ha dimensione totale 64800 bytes, ed essendo 7200 un suo divisore, occorrono 9 esecuzioni della top function per completare la cifratura di un singolo frame. Durante i test sono stati provati anche livelli di parallelismo più elevati, ma questi non hanno portato ad un aumento di prestazioni significativo come mostrato nella tabella 3.1.

La funzione *aes_cipher(plaintext, out, w)*, definita nel sorgente *aes.cpp* rappresenta la cifratura di un singolo blocco di 128 bit con l'algoritmo AES usando la chiave simmetrica cablata all'interno dell'IP Core. Per implementare questa funzione è stato utilizzato il codice sorgente C del repository Github https://github.com/dhuertas/AES.

La figura 2.8 mostra l'utilizzo delle risorse sull'FPGA a seguito della sintesi dell'IP

Core di cifratura con un livello di parallelismo pari a 450 blocchi.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	7652
FIFO	-	-	-	-
Instance	3	-	996	2025
Memory	2	-	16	2
Multiplexer	-	-	-	12200
Register	-	-	8555	-
Total	5	0	9567	21879
Available	280	220	106400	53200
Utilization (%)	1	0	8	41

Figura 2.8: Utilizzo risorse FPGA

2.2.3 Vivado design

Dopo aver completato la sintesi dell'IP Core, è necessario stabilire le interconnessioni per far comunicare l'IP Core di cifratura con il resto del sistema. Questo consente il trasferimento dei frame da cifrare e la ricezione delle immagini cifrate, pronte per essere inviate in rete. A tale scopo è stato utilizzato l'AXI DMA (*Direct Memory Access*). Il DMA è un dispositivo che permette di trasferire dati tra la memoria e le periferiche senza richiedere l'intervento del processore. Ciò consente alla CPU di non essere costantemente interrotta da un elevato numero di interrupt, consentendo invece di ricevere un solo interrupt per ogni trasferimento completato dal DMA.

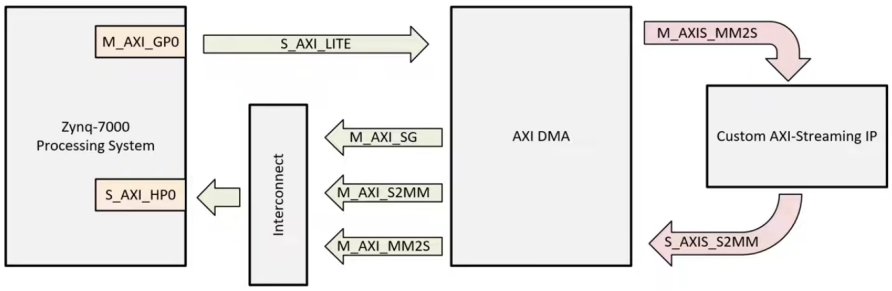

Figura 2.9: Design con Zynq, DMA e IP Core

Nel nostro progetto il DMA è impiegato per trasferire dati tra la memoria e l'IP Core di cifratura. Il processore comunica con il DMA tramite un'interfaccia AXI-lite per avviare i trasferimenti di dati tra la memoria e l'IP Core. AXI_MM2S e

AXI_S2MM sono interfacce AXI4 e forniscono al DMA l'accesso alla memoria. Le interfacce AXIS_MM2S e AXIS_S2MM sono AXI4-Stream e sono utilizzate per la comunicazione con l'IP Core.

Il design di Vivado completo è rappresentato nella figura 2.10. Da questo progetto è generato il bitstream che sarà importato dall'overlay di Python per poter comunicare con l'IP Core.

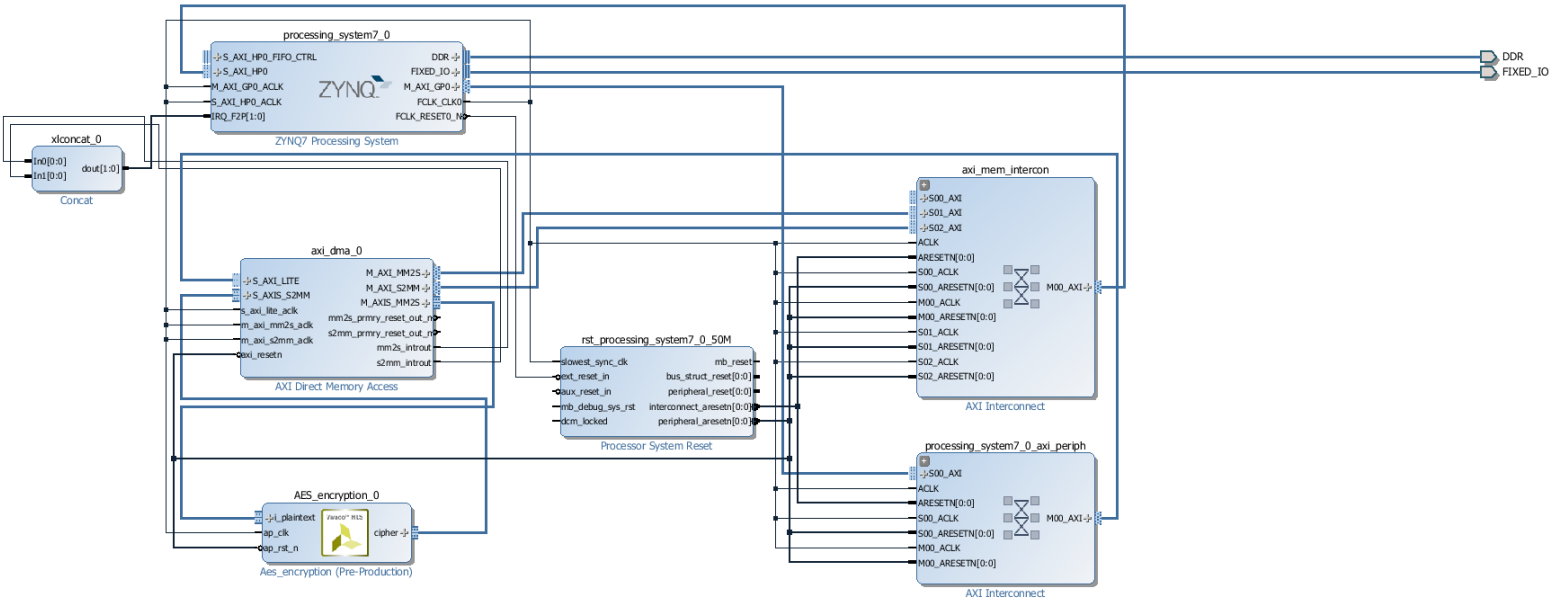

Figura 2.10: Design Vivado

2.2.4 Overlay Python

Per comunicare con l'IP Core da Python creiamo un oggetto Overlay che importa il bitstream generato da Vivado. Successivamente possiamo accede all'oggetto *dma* che useremo per comunicare con l'IP Core di cifratura.

```
1 overlay = Overlay('/home/xilinx/pynq/overlays/AES_Encryption/design_1.bit')
2
3 # Caricamento del DMA
4 dma = overlay.axi_dma_0
5
6 # Alloca memoria per l'input e l'output
7 in_buffer = pynq.allocate(shape=(data_size,), dtype=np.uint8)
8 out_buffer = pynq.allocate(shape=(data_size,), dtype=np.uint8)
```

L'oggetto *dma* dispone di un driver messo a disposizione dalla libreria pynq, che espone metodi per poter gestire i trasferimenti tra la memoria e l'IP Core in modo semplice dal punto di vista del codice.

L'immagine da cifrare viene convertita in una sequenza di byte e suddivisa in blocchi di dimensione *data_size*, impostata a 7200 (il nostro livello di parallelismo). Ogni blocco viene trasferito dalla memoria all'IP Core, cifrato e riportato in memoria. Al termine della cifratura di tutti i blocchi, i byte ottenuti vengono inseriti nel payload di un datagramma UDP e inviati al client remoto.

```
plaintext = bytearray(frame_data)

encrypted_img_bytes = b""

for i in range(0, len(plaintext), data_size):
    chunk = plaintext[i:i+data_size]

    # Copia dei dati nel buffer
    np.copyto(in_buffer, chunk)

    # Avviare il trasferimento del DMA ed aspettare i risultati
    dma.sendchannel.transfer(in_buffer)
    dma.recvchannel.transfer(out_buffer)
    dma.sendchannel.wait()
    dma.recvchannel.wait()

    encrypted_img_bytes = encrypted_img_bytes + bytes(bytearray(out_buffer))

# Invia il pacchetto al client
server_socket.sendto(encrypted_img_bytes, client_address)
```

Capitolo 3

Conclusioni

3.1 Risultati Ottenuti

Al termine dello sviluppo, l'applicazione è in grado di inviare uno stream video cifrato al client con una media di 3.6 fps, garantendo una buona fluidità e reattività dell'esperienza visiva per l'utente. Il sistema è in grado di rilevare movimenti anche impercettibili e di aprire lo streaming dopo pochi istanti. Quando il flusso inizia ad essere trasmesso, esegue senza interruzioni e termina solo dopo non aver rivelato nessun movimento per circa 5 secondi.

La cifratura di un singolo frame da 64800 bytes impiega 0.168 secondi. A livello teorico la sola cifratura limita lo stream a massimo 6 fps. Nella tabella 3.1 sono mostrati i diversi livelli di parallelismo provati durante lo sviluppo con le relative performance.

Parallelismo (blocchi)	Formato (pixel RGB)	Dimensione (B)	Tempo (s)
1	-	16	0.002
4	40x32	3840	0.05
16	56x32	5376	0.03
16	128x68	26112	0.14
256	160x128	61440	0.162
450	180x120	64800	0.168
810	180x120	64800	0.165

Tabella 3.1: Performance in base al parallelismo della cifratura

La connessione stabilita tra client e server è indipendente dalla rilevazione di movimenti, e resta stabile anche quando non viene inviata nessuna immagine. Quando il client termina, il server interrompe l'invio dello stream video, e attende una nuova connessione.

Di seguito vengono mostrate le immagini che illustrano come il sistema rileva i movimenti con la motion detection, e il confronto tra i frame registrati dalla fotocamera e i frame ricevuti e visualizzati dal client. Si nota come per ogni frame, nonostante una piccola perdita di informazione data dal resize e dalla conversione a YUV, la differenza risulti poco visibile.

Figura 3.1: maschera binaria

Figura 3.2: immagini visualizzate dal client

Inoltre, essendo ogni frame cifrato, un attaccante che tenta di decifrare il flusso video senza conoscere la chiave simmetrica usata per la cifratura, ottiene immagini totalmente incomprensibili.

Figura 3.3: decifratura con chiave sbagliata

3.2 Sviluppi Futuri

Allo stato attuale tutte le funzioni di openCV utilizzate per l'elaborazione delle immagini eseguono sul processore e impiegano dei tempi accettabili, ma con ancora margini di miglioramento.
Essendo un progetto sviluppato su Pynq, si potrebbe pensare di realizzare anche la parte hardware che esegue le funzioni di openCV e fare eseguire su FPGA anche la fase di elaborazione immagini, in modo da ottimizzare i tempi e avere prestazioni ancora più alte. Si tratterebbe quindi di realizzare un design che includa gli IP core per le funzioni di resize, conversione RGB a YUV e altre funzionalità di openCV che abbiamo usato nel codice.

Un altro sviluppo interessante che si potrebbe aggiungere è quello di realizzare l'applicazione mobile per la parte client.
In questo modo quando avviene la rilevazione dei movimenti lato server, si comunica all'utente sul suo telefono, attraverso notifica push.
Questo tipo di interazione potrebbe essere più immediata per un utilizzatore, dato che il telefono è sempre a portata di mano e potrebbe essere avvisato in qualsiasi momento.