Procesamiento de Video en Tiempo Real

Proyecto Final Procom 2023

Integrantes:

- Raimondi Marcos
- Gerard Brian
- Bean Agustin
- Arias Manuel

Tutores:

- Pola Ariel
- Leguizamon Santiago

Introducción

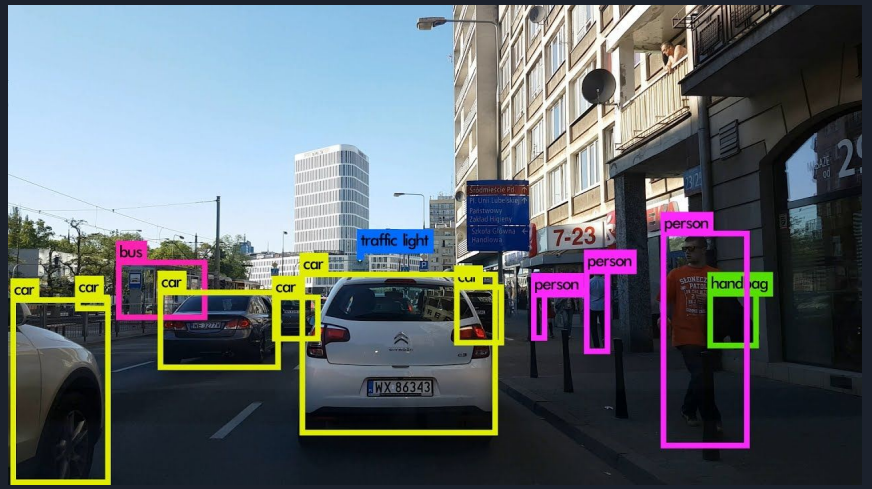
¿Qué es el procesamiento de video en tiempo real?

El procesamiento de imágenes en tiempo real es una rama de la visión por computadora que se centra en analizar y manipular imágenes digitales de forma instantánea mientras cambian.

Permite procesar información visual de manera rápida, siendo crucial en situaciones que requieren respuestas veloces.

Utiliza algoritmos y técnicas especializadas para lograr una ejecución rápida de operaciones complejas en imágenes captadas por cámaras u otros dispositivos visuales.

Tiene amplias aplicaciones, como detección de objetos, reconocimiento facial, sistemas de asistencia al conductor, realidad aumentada y monitoreo médico en tiempo real.



El Proyecto

El proyecto en sí consta de 3 etapas separadas que interactúan entre sí:

- **Aplicación Web**: es la interacción con el usuario, responsable de capturar la imagen y presentarle al usuario el resultado. El servidor captura la imagen y otro proceso la envía al microcontrolador.

- **Firmware**: es la interfaz entre el servidor web y el sistema digital, se implementan protocolos de comunicación de alta velocidad como Ethernet.

- **Program Logic**: realiza un procesamiento de alta velocidad de los frames de video que recibe. Convoluciona la imagen con un filtro de 2 dimensiones utilizando técnicas de procesamiento paralelo.

Esta división permite realizar el trabajo en paralelo, trabajando hacia interfaces y puertos que permiten levantar y probar cada etapa por separado. Con una etapa final de integración.

Diagrama en Bloques

Overview

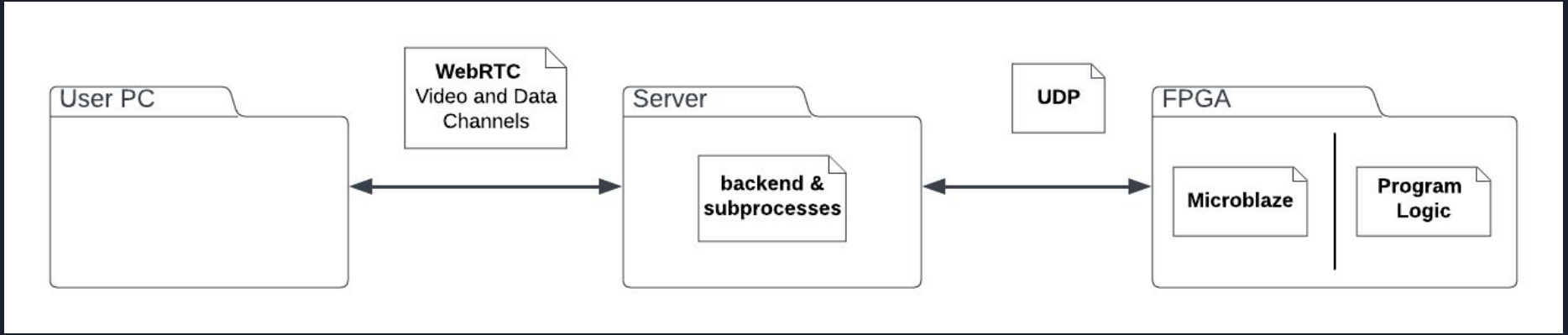


Diagrama en Bloques

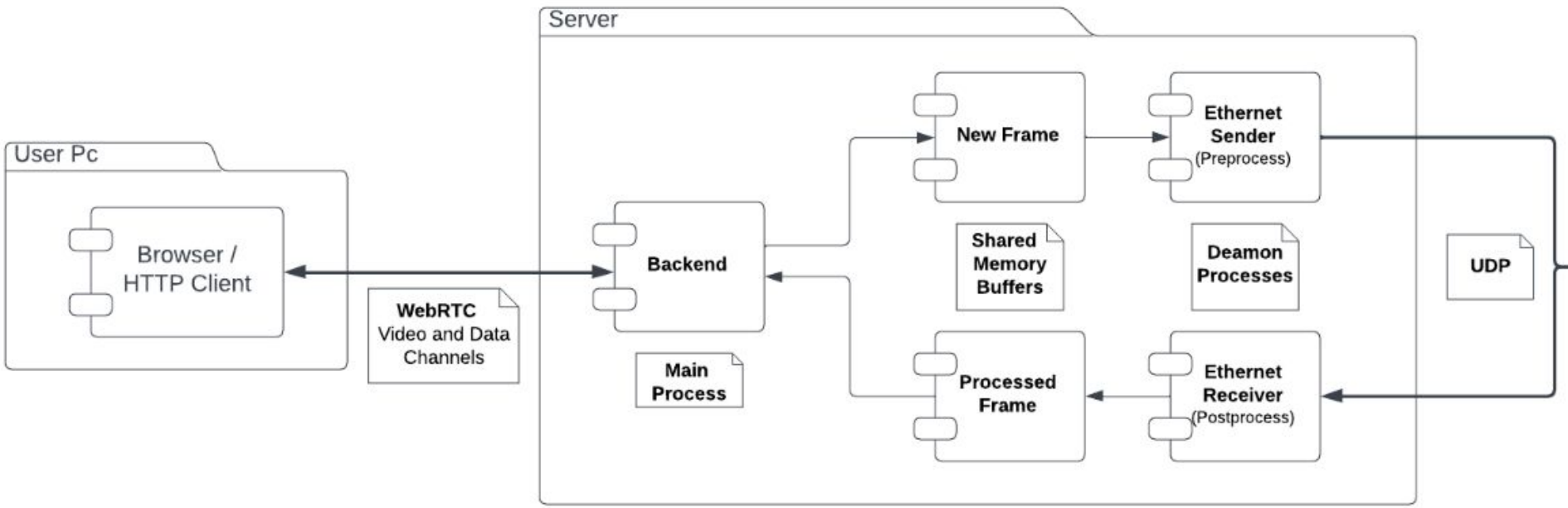
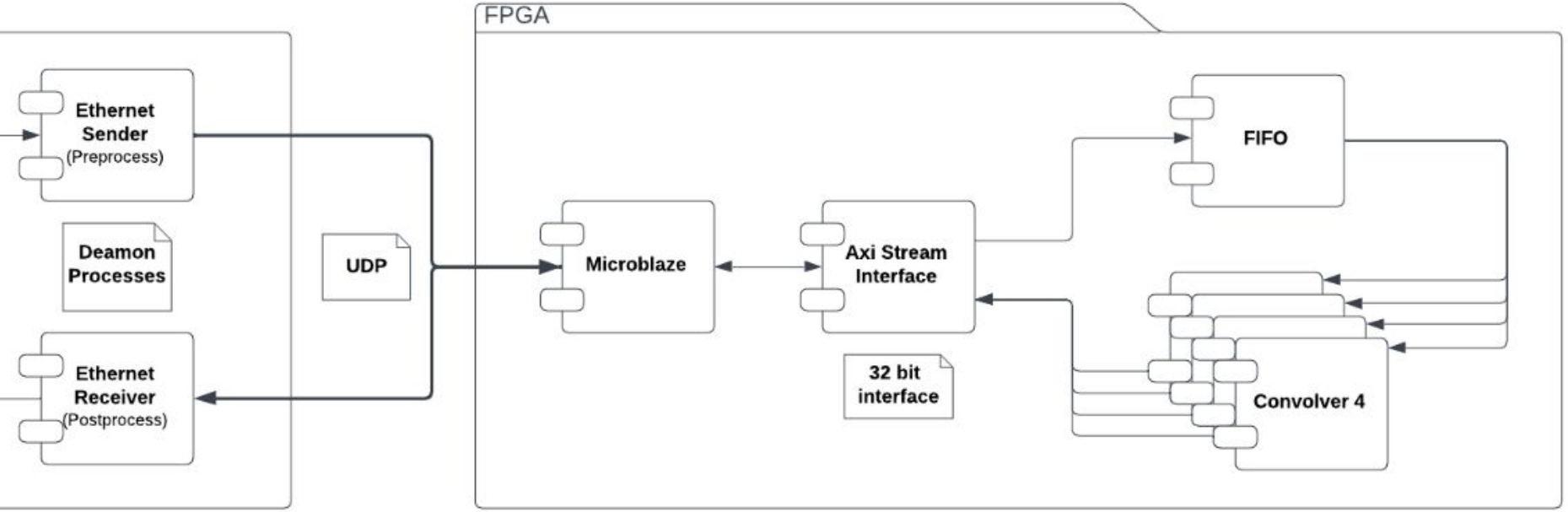
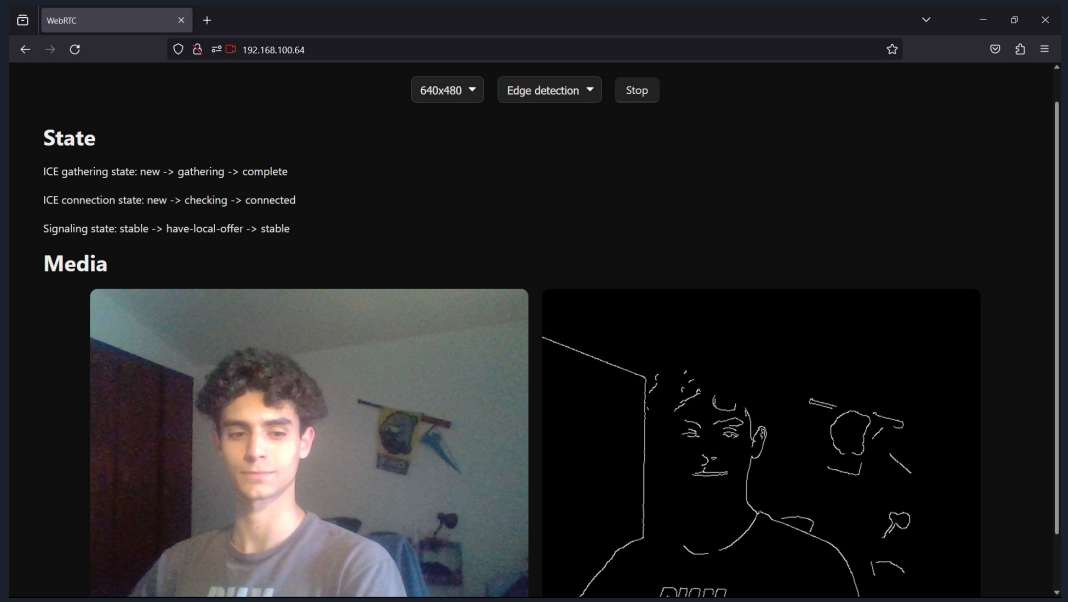


Diagrama en Bloques



La Aplicación WEB

- Frontend en javascript
- Backend en python
- Protocolo webRTC para transmitir el video
- Captura de imágen y envío de frames desde el servidor a la placa
- Pre y post procesamiento de los frames

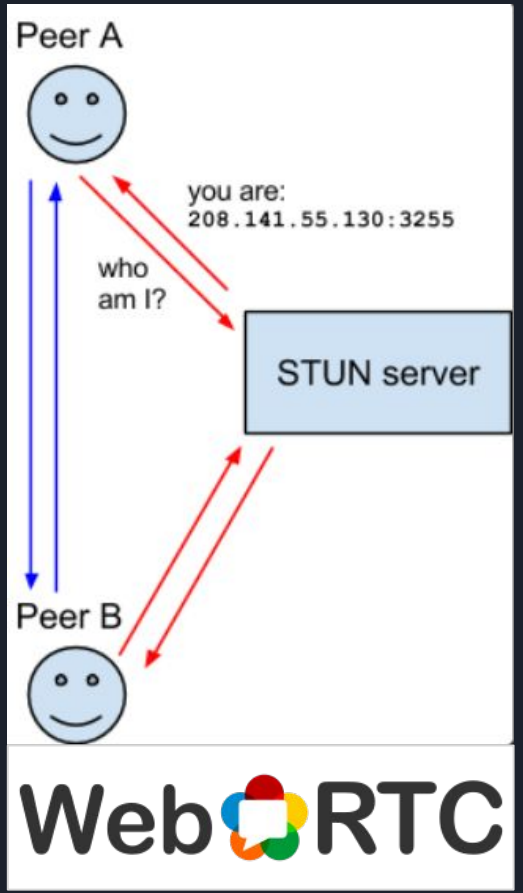


La Aplicación WEB

WebRTC: Web Real-Time Communication

- Permite comunicación en tiempo real entre navegadores web y aplicaciones
- A través de conexiones peer-to-peer
- Permite transferencia de video, audio y datos
- Sistema de Offer/Answer para establecer la línea
- ICE (Interactive Connectivity Establishment): marco que facilita la comunicación a través de un STUN server, permite establecer la conexión en redes diversas.
- Es eficiente y rápido, minimiza las conexiones (P2P)

Se establece una canal de video y otro de datos (para enviar el kernel)



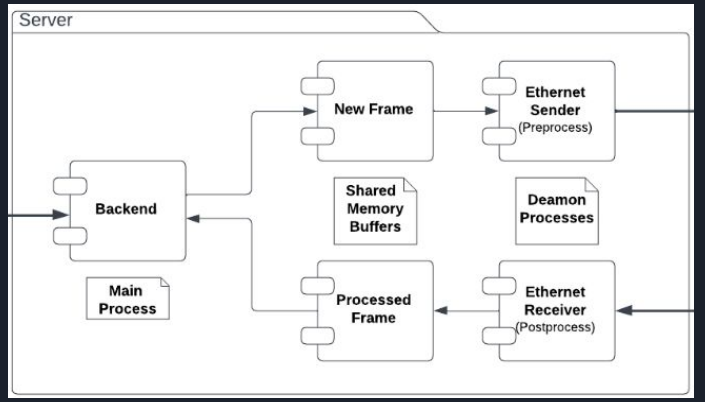
La Aplicación WEB

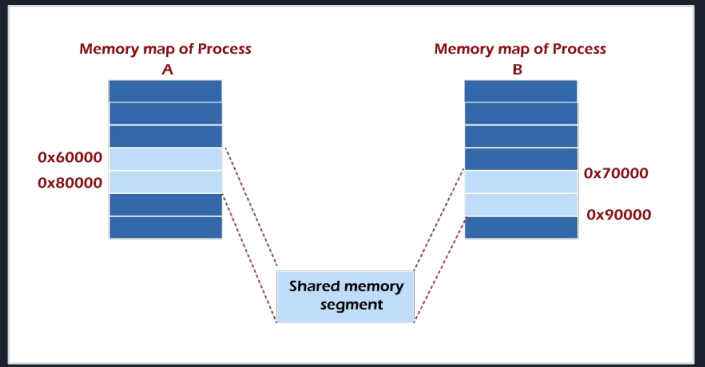
Existen 3 procesos:

- Backend web: recibe las peticiones y establece una comunicación por webRTC
- Ethernet sender: lee el último frame de la memoria compartida, lo pre procesa y lo envía al microblaze
- Ethernet receiver: recibe el frame, lo post procesa y lo escribe en al memoria compartida

Memoria compartida: para la comunicación entre procesos

- 2 buffers compartidos uno donde se coloca la imagen a procesar y otro donde se coloca la imagen procesada
- Siempre se tiene en los buffers los frames más actualizados
- Se utilizan semáforos sincronizar y para evitar condiciones de carrera entre los procesos

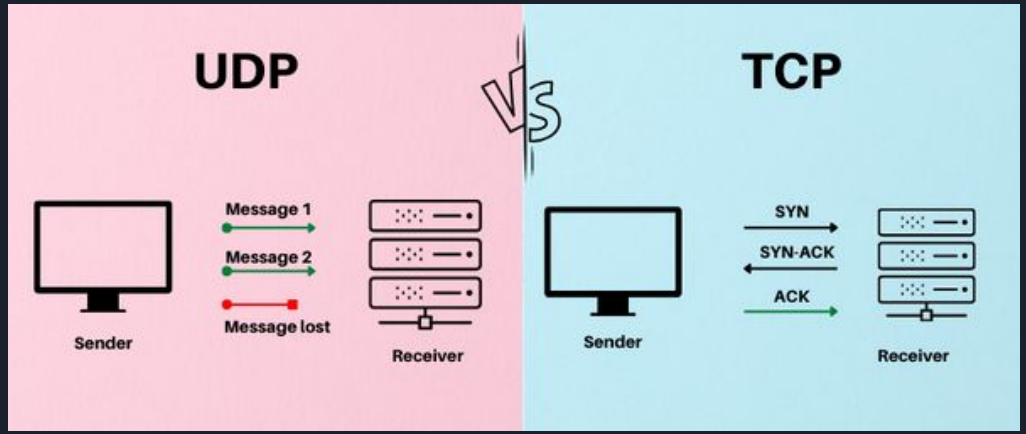




La Aplicación WEB

Ethernet: se utilizan sockets para comunicar el servidor de python con la FPGA, se debe definir el protocolo de transmisión a utilizar

- TCP: transmission control protocol, se crea una conexión y se asegura la entrega de segmentos
- UDP: user datagram protocol, no crea una conexión y no asegura la entrega, pero ahorra en overhead.

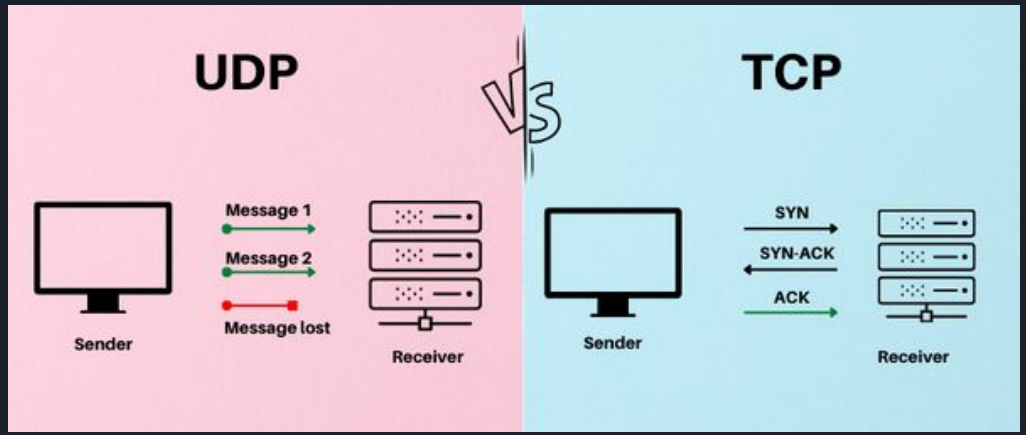


Para el streaming de video, perder un frame no es crítico, por lo que se prefieren protocolos livianos que se focalicen en la velocidad como UDP.

La Aplicación WEB

Consideraciones para utilizar UDP:

- Cada datagrama UDP que tiene un máximo de 65535 bytes
- Se reduce el frame de 640x480 a 200x200. Permite enviar un frame completo del video en cada datagrama. Facilita el procesamiento en la FPGA.
- En el server se interpola el resultado del procesamiento para devolver la imagen a su tamaño original.
- Hay un tradeoff entre velocidad y calidad.



Con TCP se puede enviar el frame original porque divide el envío en distintos paquetes que asegura que llegan y en orden.

Separar los datagramas de UDP no es una solución viable ya que el fallo de algunos invalida todo el frame.

La Aplicación WEB

Preprocesamiento:

- Se pasa la imagen a blanco y negro para trabajar en 2 dimensiones
- Se corta la imagen y redimensiona la imagen teniendo en cuenta los requerimiento en la program logic.
- Se agrega el padding
- Se ordenan los pixeles en el orden que se procesan, evitando tener que utilizar buffers para cargar todo el frame en la lógica de programa.
- Envío de metadatos (kernel y timestamp)

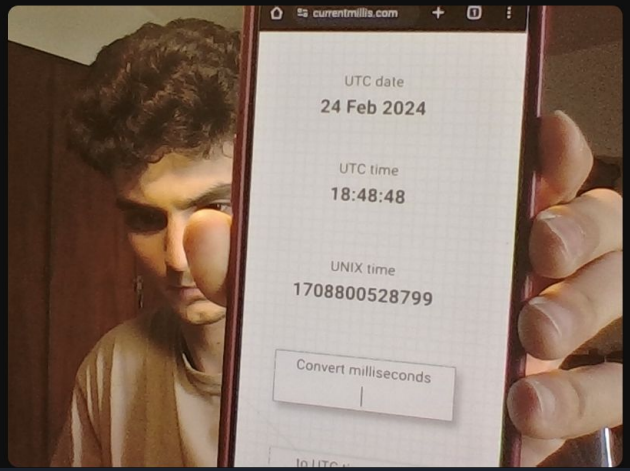
Postprocesamiento:

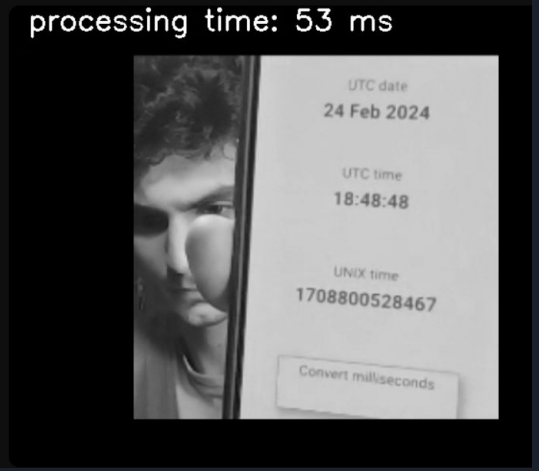
- Se toma el timestamp de los metadatos y se calcula el tiempo de procesamiento
- Se reordenan los pixeles
- Se eliminan los pixeles inválidos (de los transitorios, primeras dos filas y columnas)
- Se redimensiona la imagen al tamaño original

La Aplicación WEB

Desde el webserver se puede realizar control de flujo de los frames, enviando un frame nuevo cada cierto tiempo para no sobrecargar la placa. Tambien se podria bajar la resolucion para enviar frames más pequeños. Se obtuvo una buena relación calidad y performance con frames de 200x200 cada 0.07 segundos.

Media





- Tiempo de procesamiento: 53 ms
- Tiempo de retorno: 332 ms

La Aplicación WEB

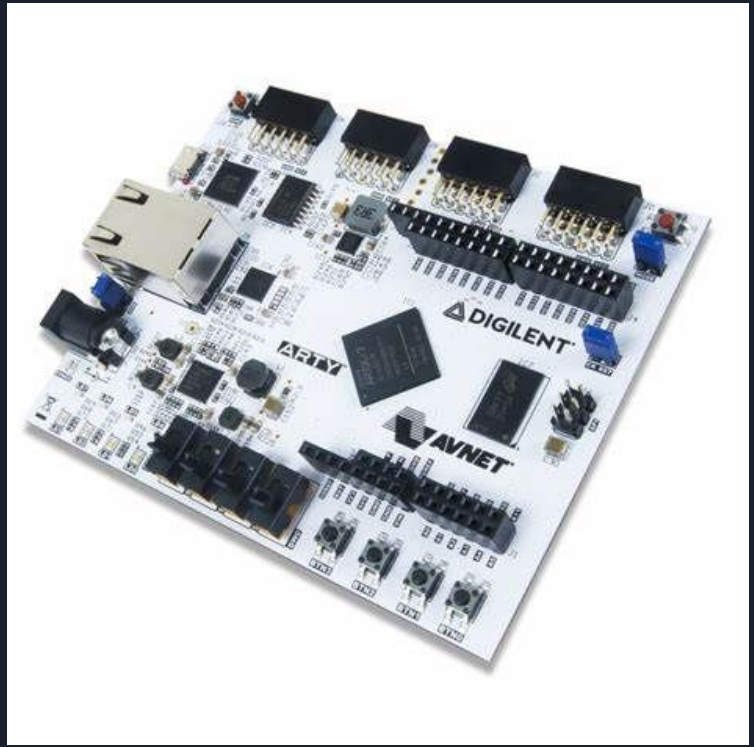
Algunos resultados obtenidos a lo largo del desarrollo:

#	Protocolo	Red	Retorno al server (ms)	Retorno al usuario (ms)
1	TCP	localhost	1.86	200
2	TCP	local network	24.30	200
3	TCP	vpn fpga	41.51	210
4	UDP	localhost	0.68	160
5	UDP	vpn fpga	50~60	332

Solo en #5 se agregan los pasos pre y post procesamiento en el server. En #3 la fpga solo devolvía el frame sin procesarlo. En el resto de los casos se simulaba con un script de python.

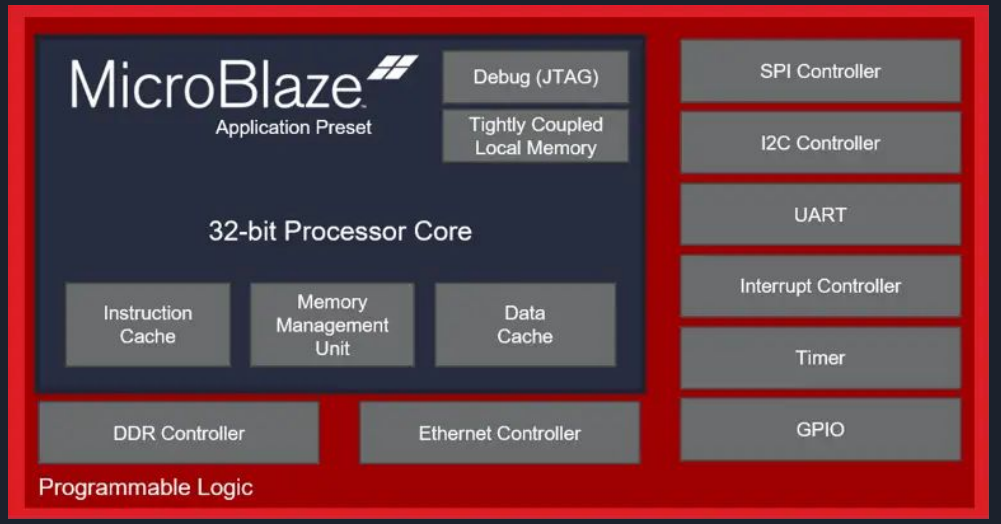
FPGA

- Field-Programmable Gate Arrays.
- Utilizan lenguajes de descripción de hardware, como Verilog.
- Gran flexibilidad y reusabilidad.
- Diseño rápido y bajo tiempo al mercado.
- Permite probar implementaciones antes de pasar a un circuito integrado de aplicación específica (ASIC).
- Equilibrio entre rendimiento y flexibilidad.
- Modelo utilizado: Arty A7-35.



Microblaze

1. Microprocesador suave (Soft Processor). Se puede implementar completamente mediante síntesis lógica.
    - Arquitectura Harvard.
    - Conjunto de instrucciones de 32-bit.
2. Brinda mucha flexibilidad en uso para los usuarios, dependiendo de la aplicación que se le pida.
    - Microcontroller.
    - Real- Time Processor.
    - Application Processor (Admite linux integrado).
3. Tiene soporte para la familia Artix-7.

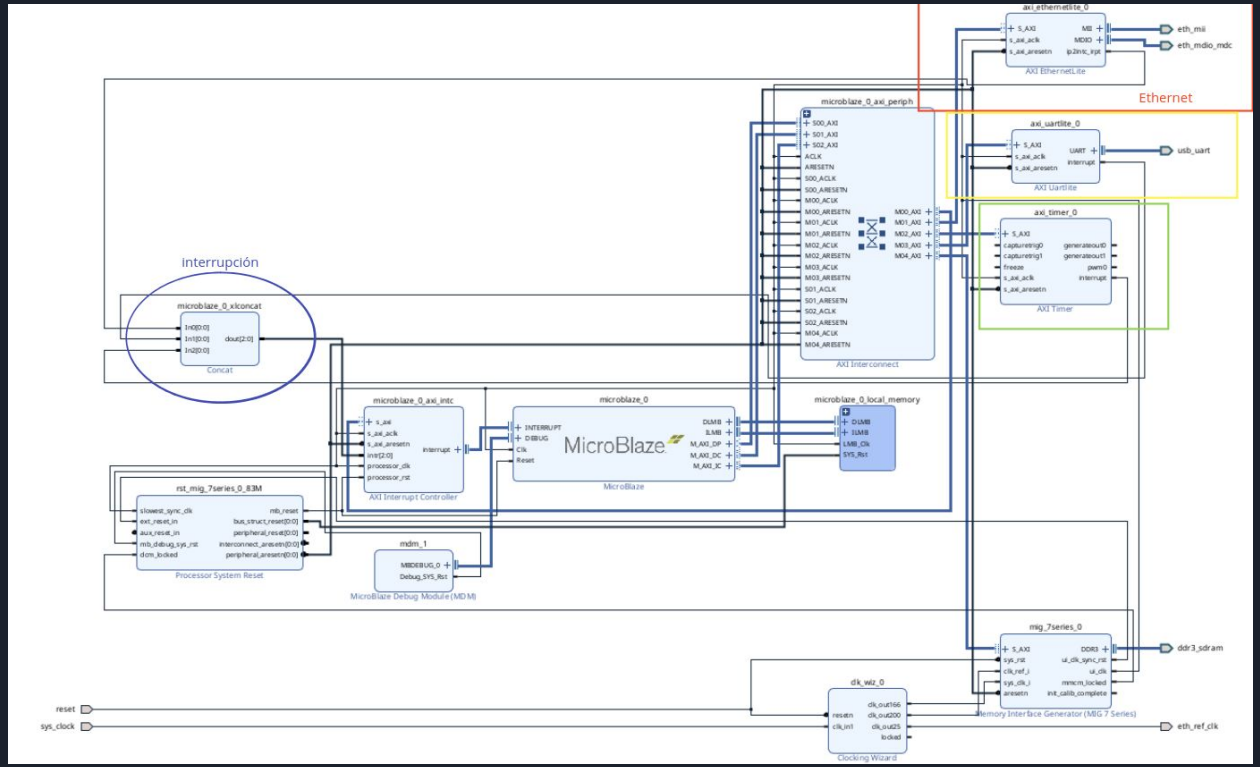


Datasheet

Primeros pasos Ethernet: echo server-tcp

Como primeros pasos implementamos un servidor echo server para poder probar:

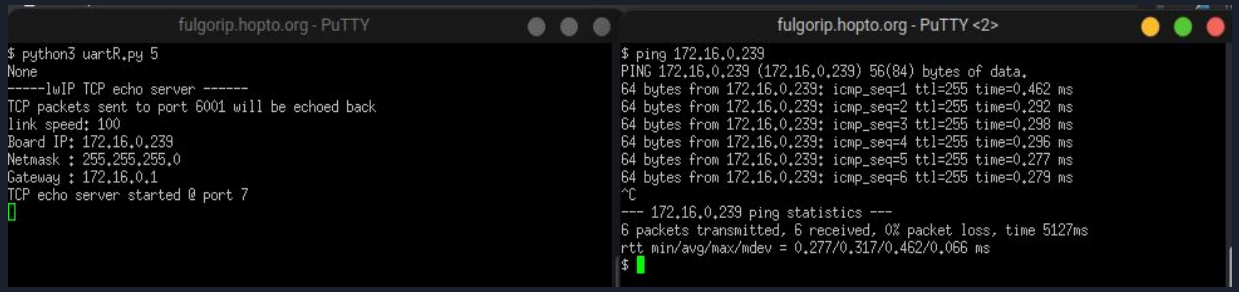
- Los IP core a usar: ethernet_lite, axi_timer, uart_lite.
- El microblaze.
- Cómo reacciona la fpga, ¿responde?



Pruebas: echo server

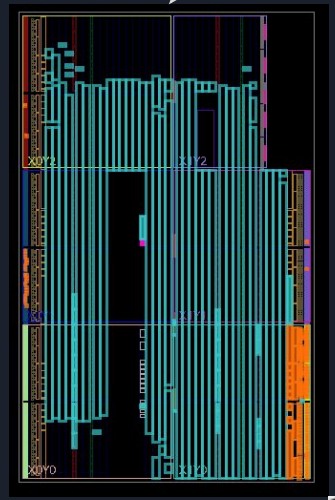
- ✓ Microblaze
- ✓ Uart
- ✓ Ethernet

Prueba de ping en la placa



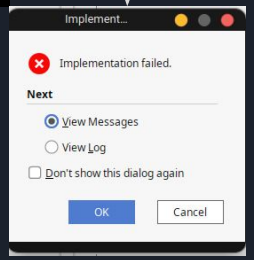
Envio y recepcion de algunos datos

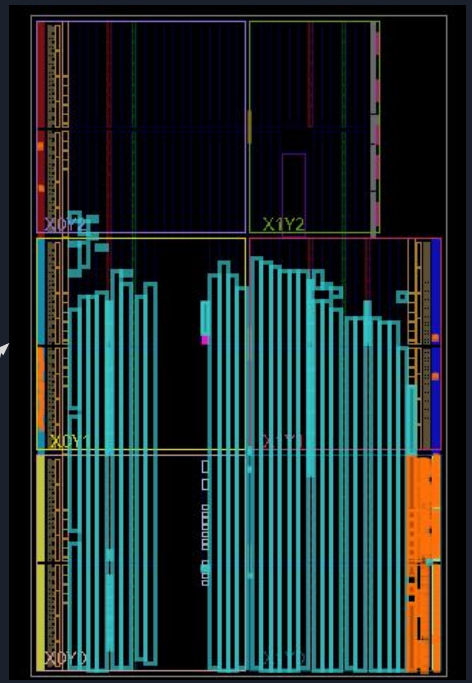
Observaciones



Se intentó utilizar el IP core AXI cdma que incluye las funcionalidades de Ethernet y DMA, sin embargo ocupaba mucho espacio y no se logró su implementación en la FPGA (ocupa más lut de las que tiene la Arty A7-35).

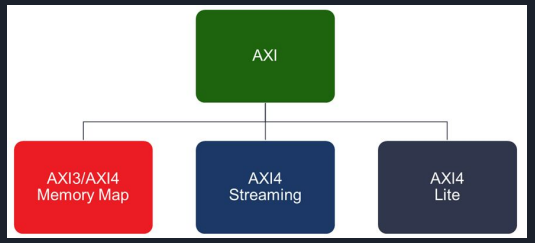
Se opta por tomar ethernet sin dma ocupando menos espacio y logrando su implementación.

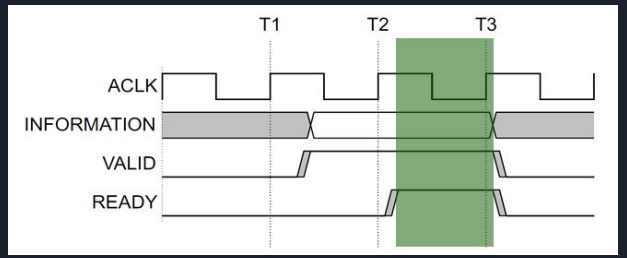




Axi Stream

- Microblaze soporta AXI.
- AXI, que significa **I**nterfaz e**X**tensible **A**vanzada, es un protocolo de interfaz definido por ARM como parte del estándar AMBA (Advanced Microcontroller Bus Architecture).
  - AXI 4: Para alta performance de requerimientos memory-mapped.
  - AXI4-Lite: para comunicación memory-mapped simple y de bajo rendimiento.
  - **AXI4-Stream**: Para alta velocidad de transmisión de datos.
- Bus de datos de **32 bits**.
- La señal **Valid** informa que el Master está enviando un dato válido.
- La señal **Ready** proveniente del slave indica que está preparado para recibir un nuevo dato.
- La transacción de información se realiza cuando el dato es válido y se está listo para recibir (**Valid && Ready**)





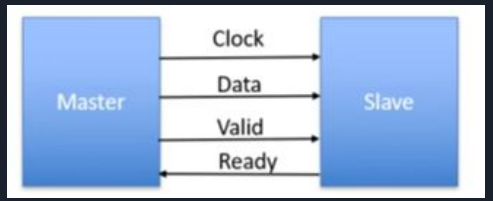
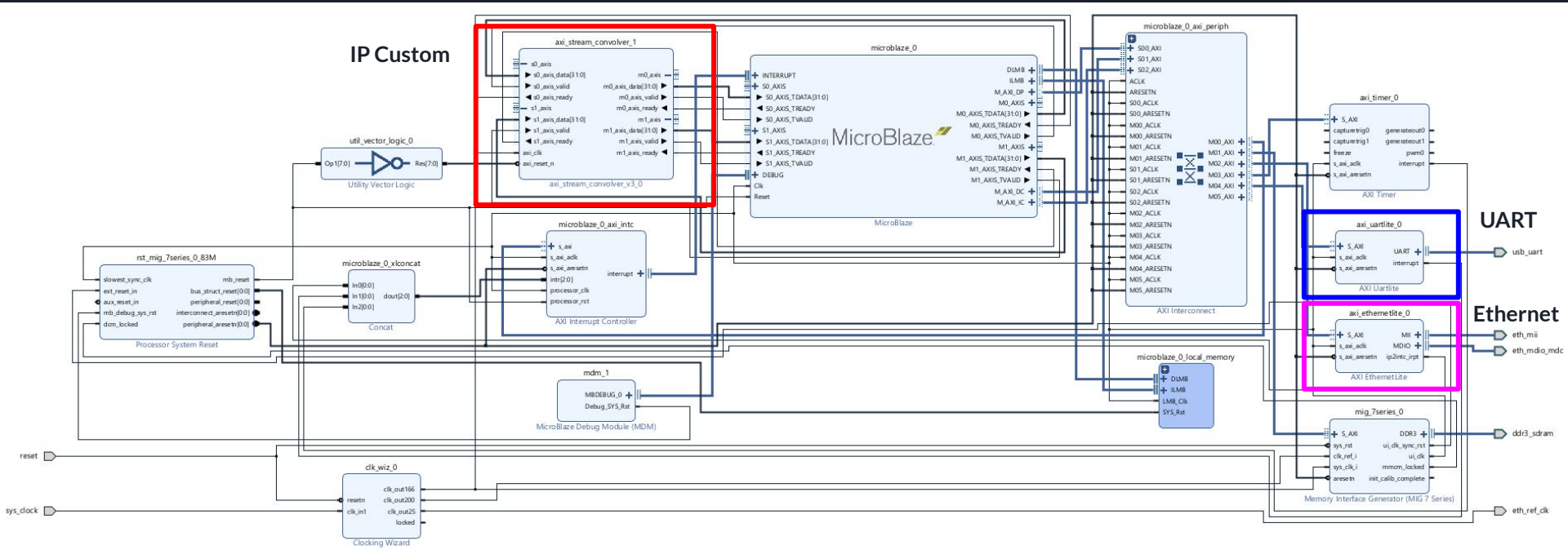


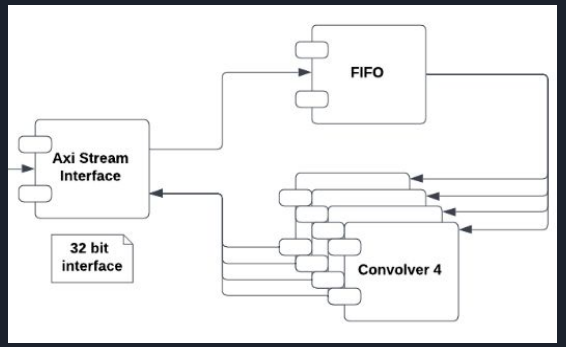
Diagrama de Bloques

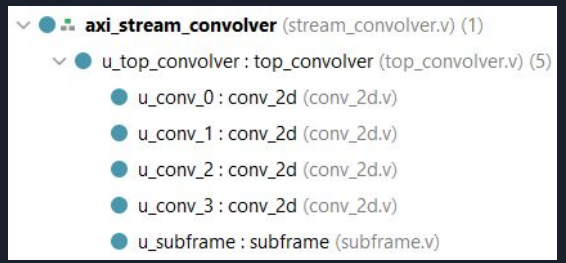
Se agrega el módulo UART para facilitar el debugging del microblaze.

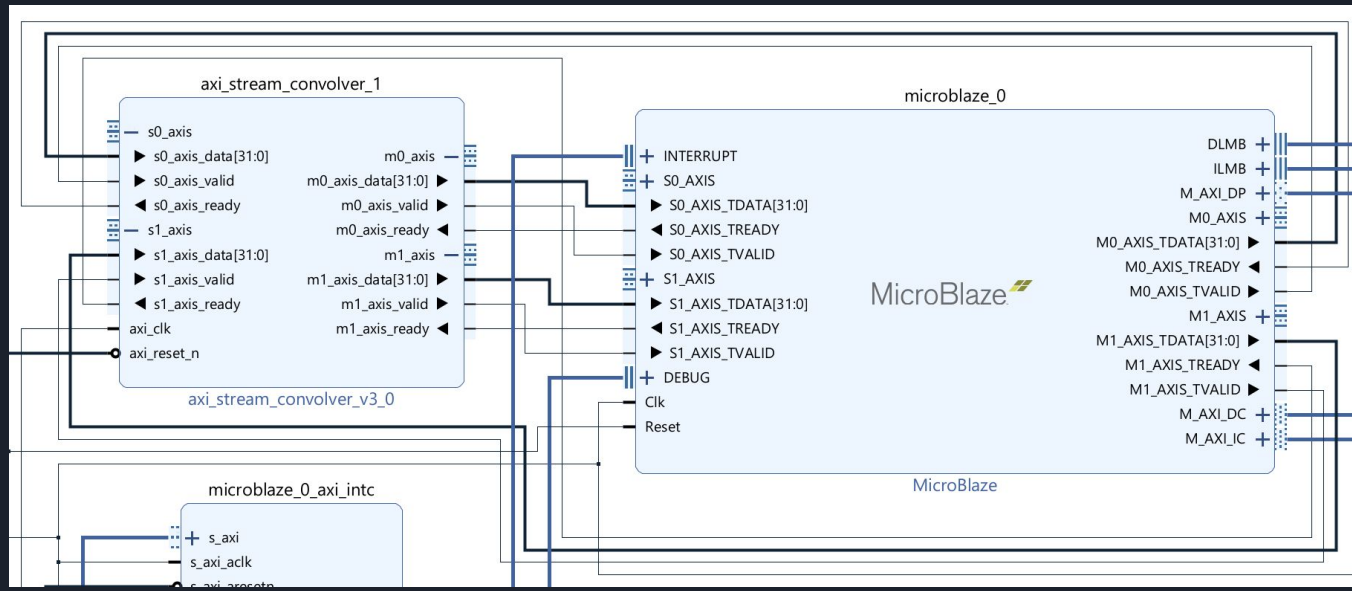


Custom IP

Debido a que la interfaz AXI Stream en microblaze es de 32bits, se busca un sistema que permita convolucionar de a 4 píxeles por ciclo de reloj.





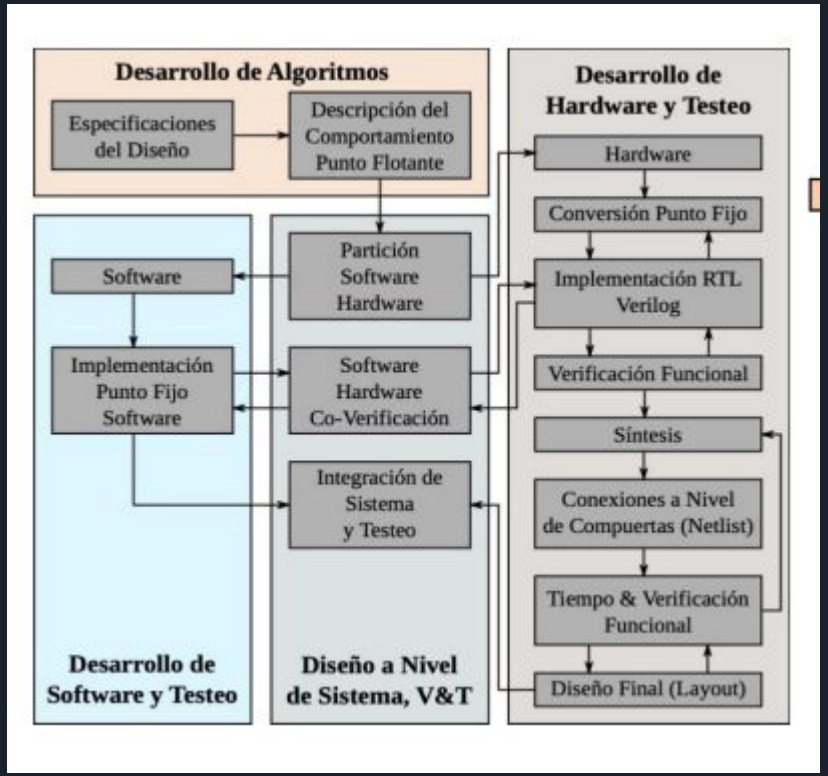


Program Logic

Flujo de Diseño de Hardware

Las etapas de diseño más importantes son:

- ➔ Comportamiento en punto flotante en Python.
- ➔ Implementación en punto fijo en Python.
- ➔ Implementación en Verilog y testing.
- ➔ Implementación en FPGA.



Objetivos del Diseño

Paralelismo

Diseño de circuito para realizar múltiples operaciones de convolución simultáneamente, para mejor rendimiento y alcanzar los requerimientos de velocidad necesarios.

E/S

Definir cómo se ingresarán los datos de la imagen y el kernel a la FPGA y cómo se leerán los resultados de la convolución.
Hay diferentes opciones: *DMA*, *GPIO*, *AXI Stream*.

Kernel

Implementar kernels que permitan realizar diferentes filtrados para obtener resultados distintos como imagenes en blanco y negro, con detección de borde o con difuminado.

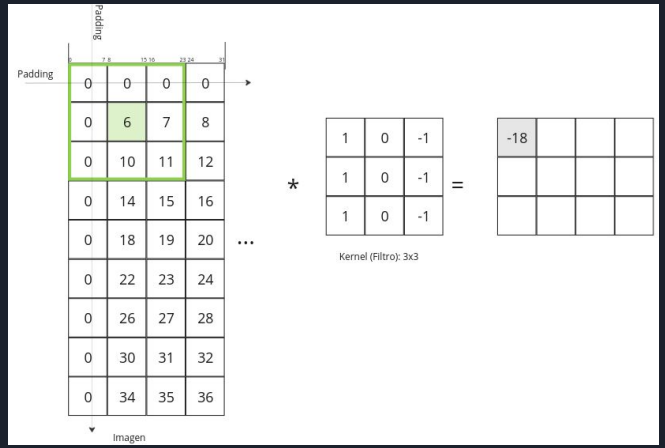
Convolución 2D

- ➔ Se cuenta con una **Imagen f(x,y)** de *(m x n)* píxeles.
- ➔ El **Kernel *K(x,y)*** tiene un tamaño *(k x k)* que es definido por el filtro que se quiere utilizar.
- ➔ La **imagen** deseada **g(x,y)** es el resultado de una convolución bidimensional, definida en este caso como:

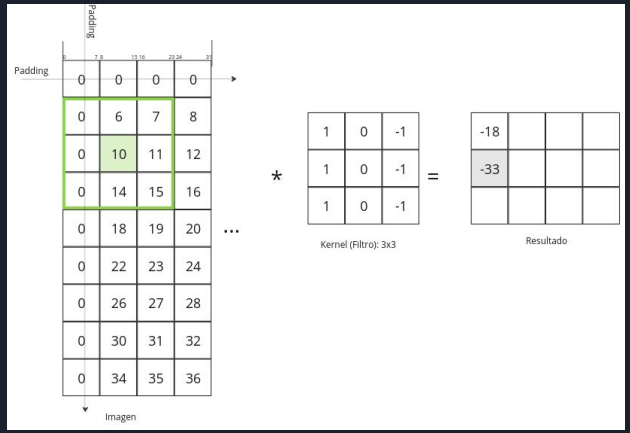
$$g(x,y) = \sum_{i=0}^{2} \sum_{j=0}^{2} f(i,j) \cdot K(x-i, y-j)$$

Convolución 2D

- El primer paso es definir el kernel, que es una matriz que especifica los pesos de la operación de convolución.
- En los bordes se debe agregar un “padding” para poder realizar la convolución. En este caso se elige rellenar con 0s.
- Con un kernel 3x3 se deben procesar 9 datos al mismo tiempo para obtener un píxel.



Primera iteración

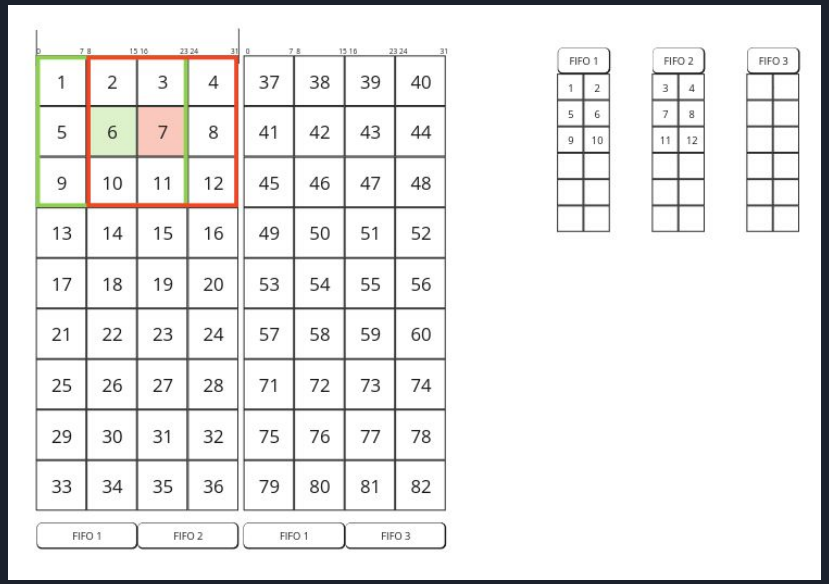


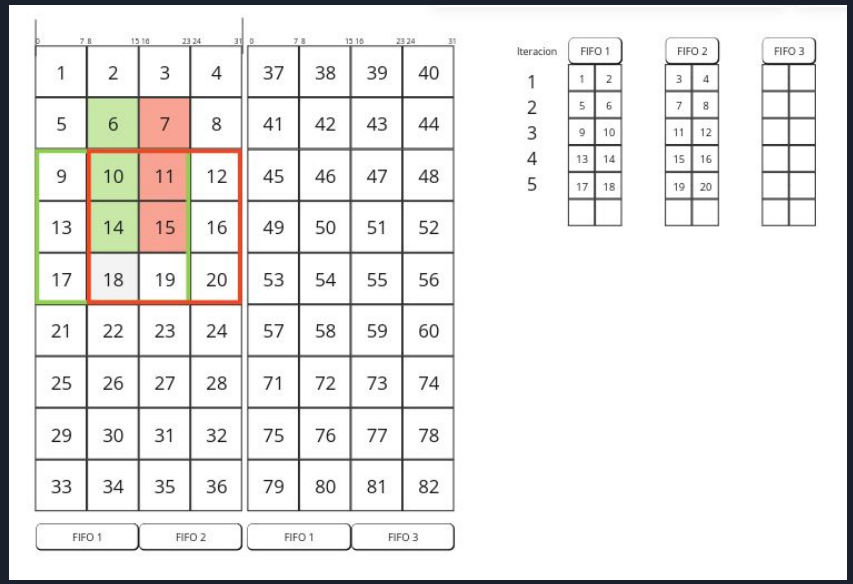
Segunda iteración

Arquitectura

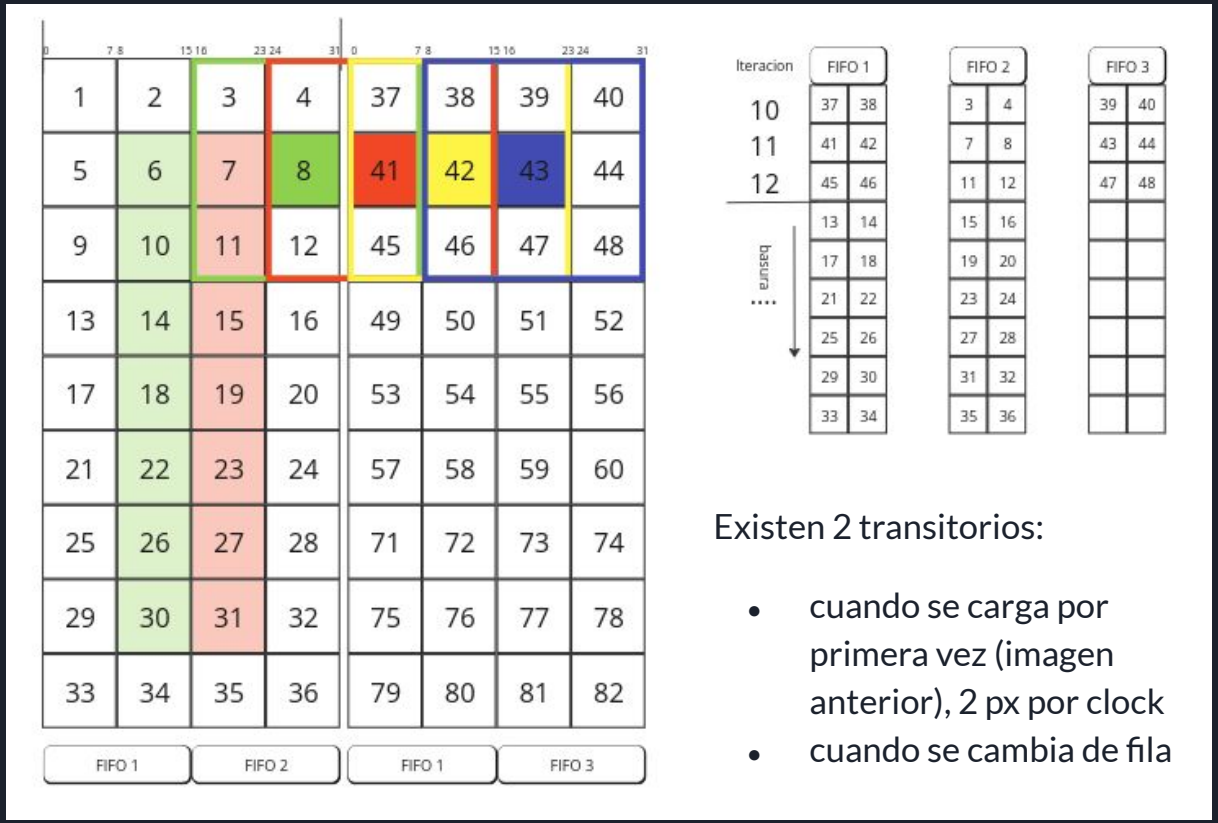
Primera aproximación

4 convolucionadores en paralelo, usando 3 fifos





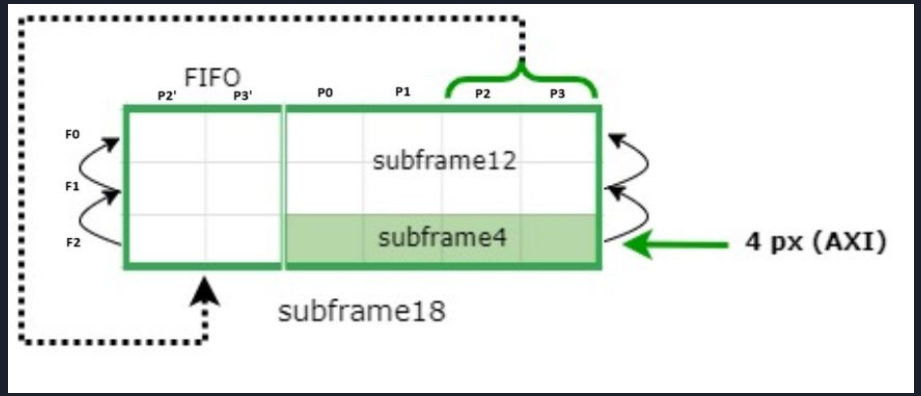
➔ Cada pixel se lee una sola vez y se obtiene un resultado (4 píxeles) por clock



➔ Tiene el inconveniente de que se guardan valores que no se utilizan nuevamente (valores de la fifo 1)

Arquitectura

Diseño final



x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

- ➔ Disminuye el hardware necesario: utiliza una sola FIFO y un buffer de 12 pixeles.
- ➔ Solo se guardan los pixeles útiles.
- ➔ Se tiene en cuenta que en cada ciclo se obtienen 4 pixeles y se devuelven otros 4, por lo que no hacen falta buffers intermedios para la transmisión.

Arquitectura

Caso especial: primeras filas

x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

Arquitectura

Caso especial: primeras columnas

x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

Arquitectura

Primeras columnas

x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

Arquitectura

Resto de columnas

x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

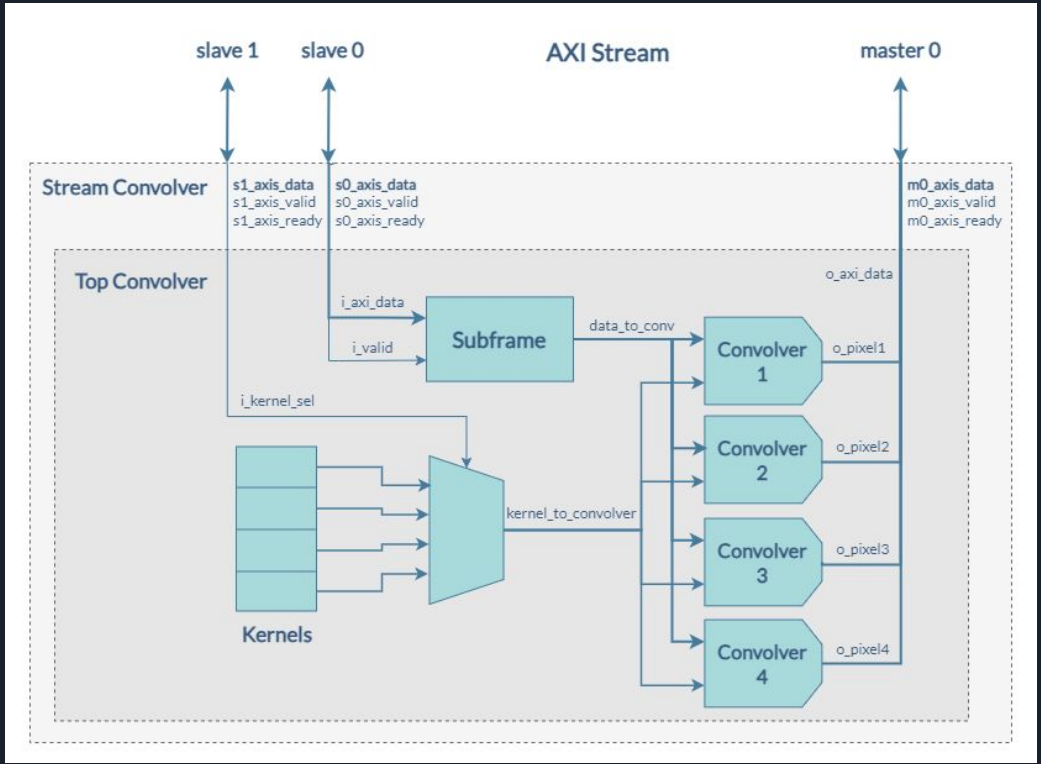
Arquitectura

Resto de columnas

x	x	0	0	0	0	0	0	0	0	0	0	0	0
x	x	0	0	1	2	3	4	5	6	7	8	9	0
x	x	0	10	11	12	13	14	15	16	17	18	19	0
x	x	0	20	21	22	23	24	25	26	27	28	29	0
x	x	0	30	31	32	33	34	35	36	37	38	39	0
x	x	0	40	41	42	43	44	45	46	47	48	49	0
x	x	0	50	51	52	53	54	55	56	57	58	59	0
x	x	0	60	61	62	63	64	65	66	67	68	69	0
x	x	0	70	71	72	73	74	75	76	77	78	79	0
x	x	0	80	81	82	83	84	85	86	87	88	89	0
x	x	0	90	91	92	93	94	95	96	97	98	99	0
x	x	0	0	0	0	0	0	0	0	0	0	0	0

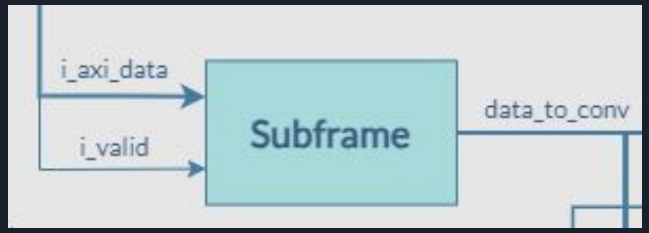
Arquitectura

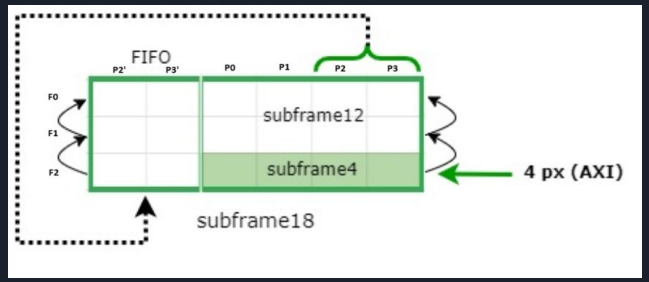
Implementación en Hardware

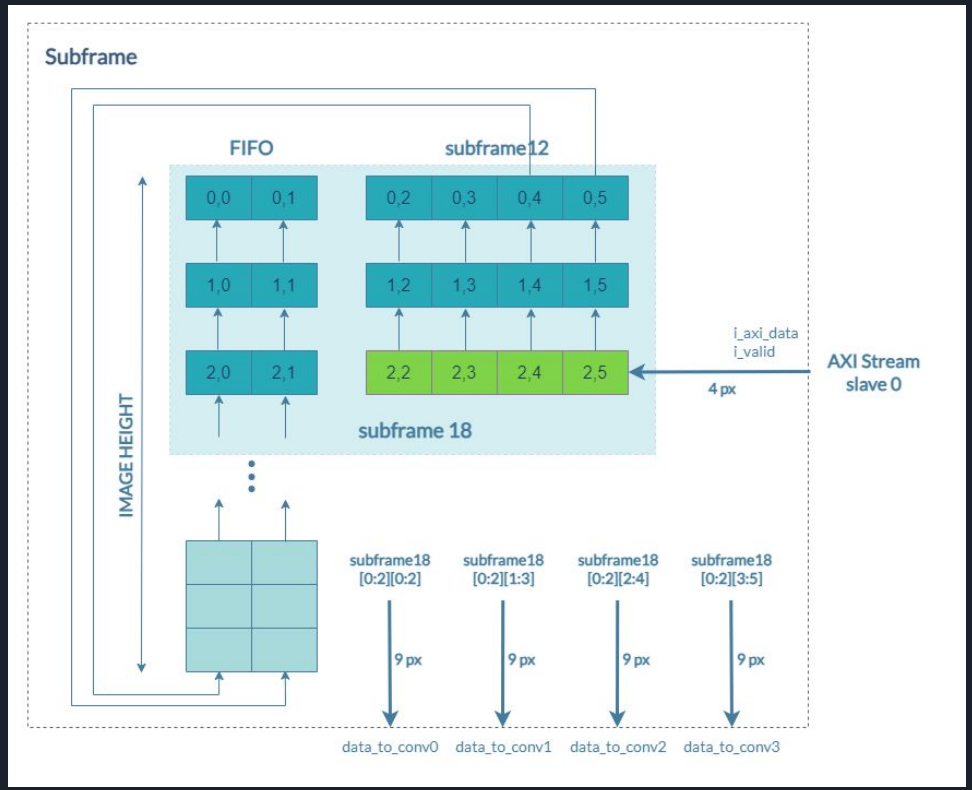


Arquitectura

Implementación en Hardware

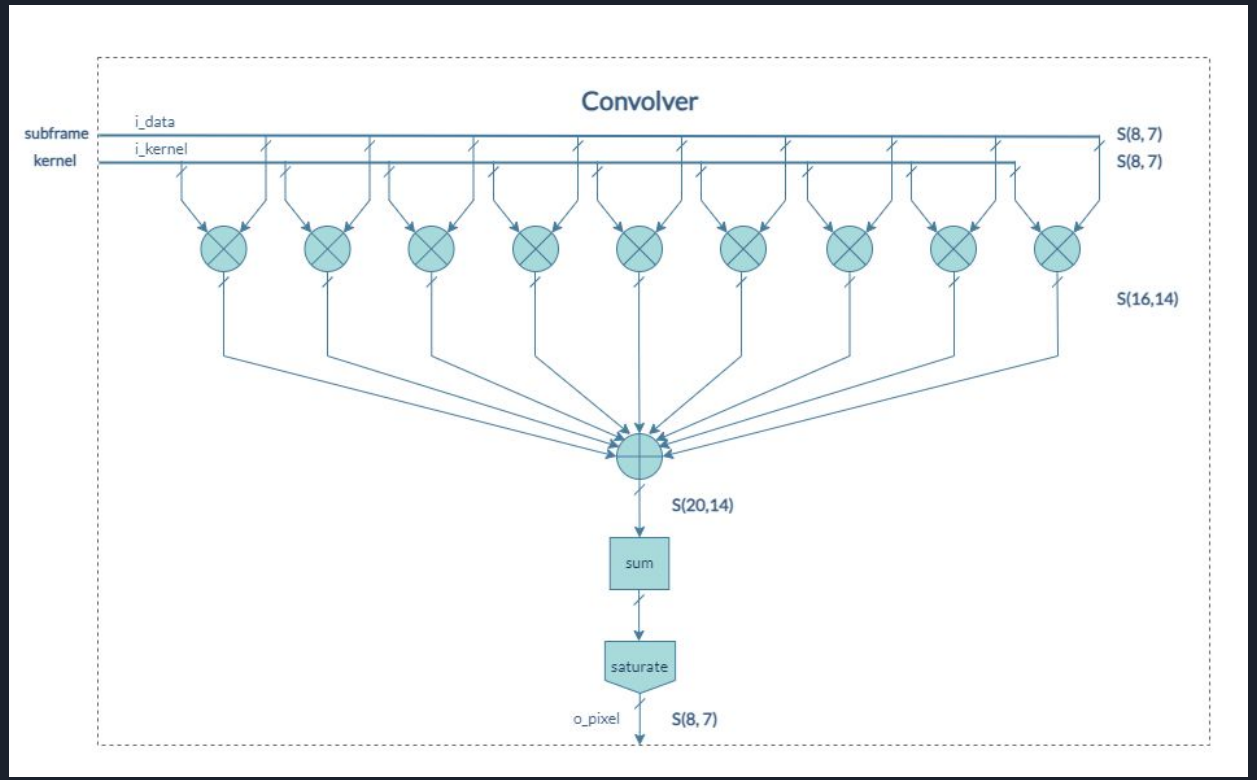






Arquitectura

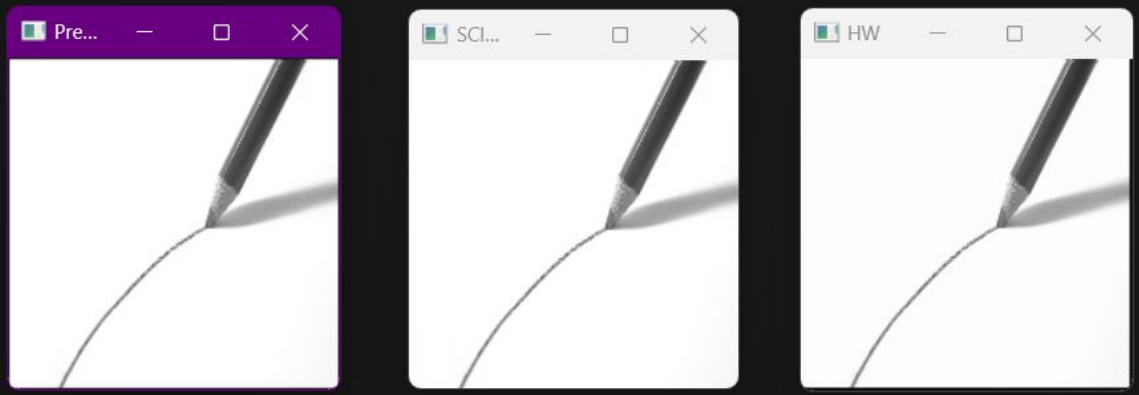
Implementación en Hardware



Resultados de simulación en Python

Kernel identidad

```
processed_hw
[[253 253 253 ... 254   0   0]
 [253 253 253 ... 254   0   0]
 [253 253 253 ... 254   0   0]
 ...
 [255 255 255 ... 246   0   0]
 [  0   0   0 ...   0   0   0]
 [  0   0   0 ...   0   0   0]]
processed
[[255 255 255 ... 254 254 254]
 [255 255 255 ... 254 254 254]
 [255 255 255 ... 254 254 254]
 ...
 [255 255 255 ... 246 246 246]
 [255 255 255 ... 246 246 246]
 [255 255 255 ... 246 246 246]]
```



Preprocesada

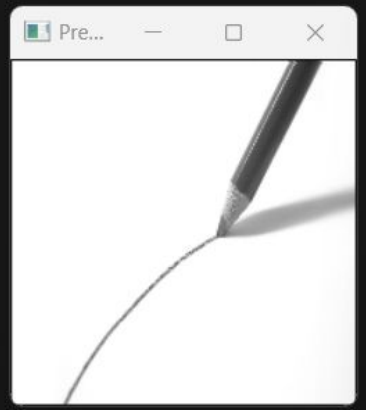
Convolución con scipy

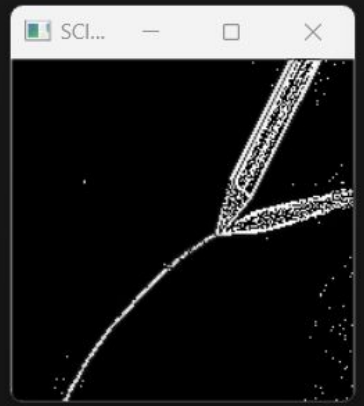
Convolución con algoritmo propio

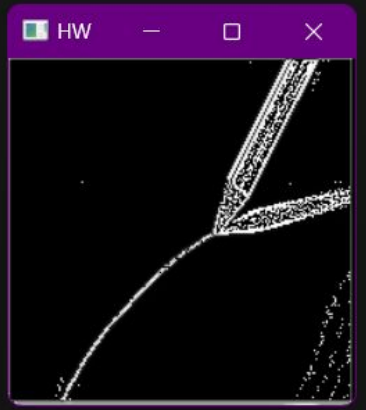
Resultados de simulación en Python

Kernel detector de bordes

```
processed_hw
[[158  94  94 ... 159   0   0]
 [ 94   0   0 ...  95   0   0]
 [ 94   0   0 ...  95   0   0]
 ...
 [159  96  96 ... 153   0   0]
 [  0   0   0 ... 164   0   0]
 [  0   0 162 ... 193   0   0]]
processed
[[159  95  95 ...  95  95 158]
 [ 95   0   0 ...   0   0  95]
 [ 95   0   0 ...   0   0  95]
 ...
 [ 95   0   0 ...   0   0  92]
 [ 95   0   0 ...   0   0  92]
 [159  95  95 ...  91  92 153]]
```







Preprocesada

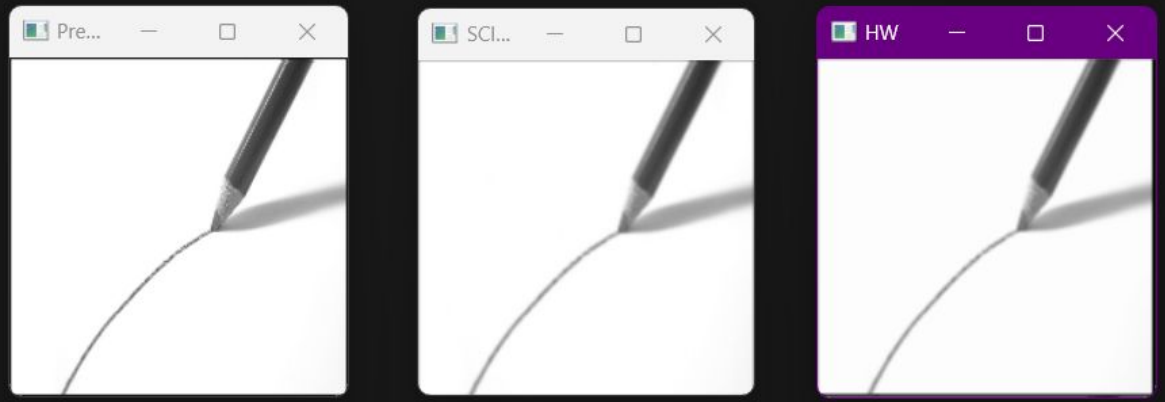
Convolución con scipy

Convolución con algoritmo propio

Resultados de simulación en Python

Kernel gaussiano

```
processed_hw
[[142 189 189 ... 142   0   0]
 [189 253 253 ... 190   0   0]
 [189 253 253 ... 190   0   0]
 ...
 [143 190 190 ... 138   0   0]
 [  0   0   0 ...  61   0   0]
 [  0   0  63 ...  47   0   0]]
processed
[[143 191 191 ... 190 190 142]
 [191 255 255 ... 254 254 190]
 [191 255 255 ... 254 254 190]
 ...
 [191 255 255 ... 246 246 184]
 [191 255 255 ... 246 246 184]
 [143 191 191 ... 184 184 138]]
```



Preprocesada

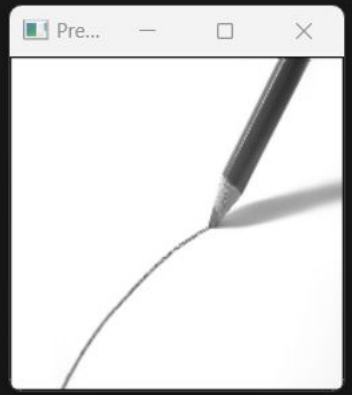
Convolución con scipy

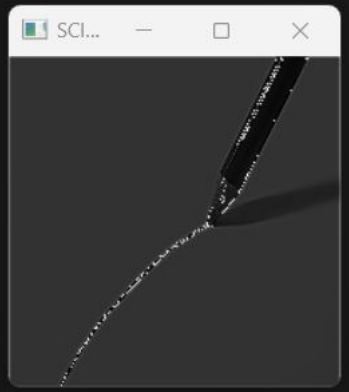
Convolución con algoritmo propio

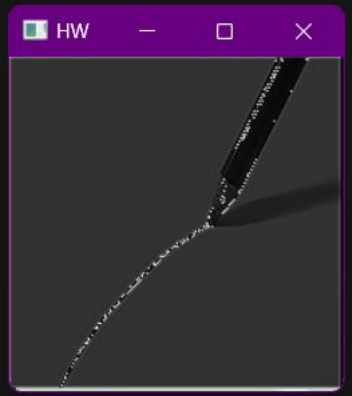
Resultados de simulación en Python

Kernel sharpen

```
processed_hw
[[151 101 101 ... 152   0   0]
 [101  50  50 ... 101   0   0]
 [101  50  50 ... 101   0   0]
 ...
 [153 102 102 ... 147   0   0]
 [  0   0   0 ... 207   0   0]
 [  0   0 206 ... 206   0   0]]
processed
[[153 101 101 ... 101 101 152]
 [101  50  50 ...  50  50 101]
 [101  50  50 ...  50  50 101]
 ...
 [101  50  50 ...  49  48  98]
 [101  50  50 ...  49  49  98]
 [153 102 102 ...  98  98 147]]
```





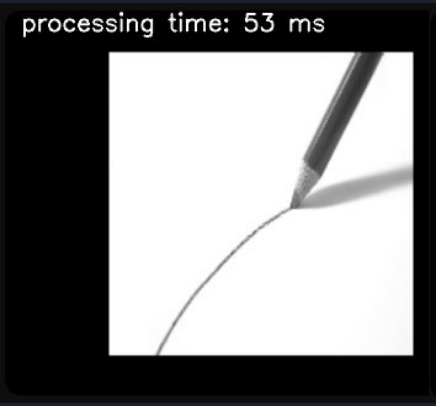


Preprocesada

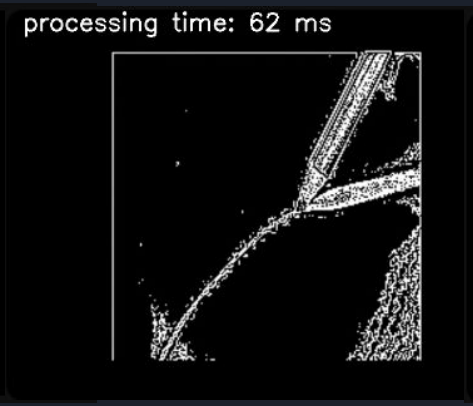
Convolución con scipy

Convolución con algoritmo propio

Resultados FPGA

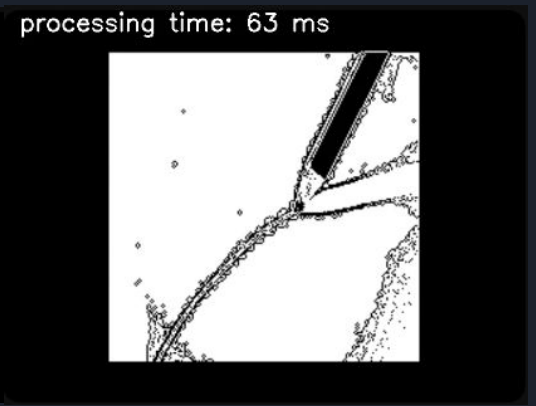


Identidad



Edges

Gaussian Blur



Sharpen

🔥🔥💀💀 Demo en VIVO 💀💀🔥🔥

🏁 FIN 🏁

Roadmap

Web Server:

- ✓ iniciar ejemplo webrtc
- ✓ ajustar ejemplo (quitar codigo que no sirve, refactorizar)
- ✓ crear proceso de comunicación ethernet
- ✓ crear contenedor de docker (Dockerfile, docker-compose)
- ✓ crear servicio en python de procesamiento de video
- ✓ agregar funcionalidad de sockets TCP
- ✓ utilizar UDP
- ✓ probar funcionalidad en localhost y en pcs diferentes
- ✓ acceder a la vpn y al server de la funda por ssh
- ✓ levantar servicios en el server de la funda
- ✓ se divide el proceso Ethernet en 2, uno para enviar y otro para recibir.
- ✓ integrar con microblaze con UDP (TCP ya funciono)
  - ✓ se logra con frame 50x50 y mas grande
  - ✓ evaluar cambio de resolución dependiendo del desempeño
  - ✓ analizar orden de pixeles (para facilitar procesamiento)
  - ✓ hacer el preprocesamiento en el server (ordenar los pixeles)
  - ✓ hacer el postprocesamiento (reordenar los pixeles en el frame)
- ✓ cambiar kernel durante la transmision

Microblaze:

- ✓ Funciona ethernet con tcp

  X No se pudo implementar el ip core de Axi dma.

- ✓ Funcionando ethernet con UDP
  - ✓ problemas para recibir paquetes grandes de UDP
- ✓ analizar cantidad de memoria disponible si se instancio ddr
  - ✓ se instancio cantidad suficiente
- ✓ evaluar funcionamiento con gpio hacer loopback
  - ✓ escribir y leer una matriz en la bram por gpio (en progreso)
- ✓ ver opciones de ips para comunicación con hardware:
  - bram controller con memory generator
  - ✓ interfaz stream link, (habilitar canal de alta velocidad en microblaze)
  - ipcore con interfaz Axi comun
- ✓ crear ip custom que implemente la interfaz AXI STREAM
- ✓ integrar SERVER + ETHERNET + STREAM:
  - ✓ hacer loopback por la interfaz de axi stream
  - ✓ enviar y recibir metadatos

Program Logic:

- ✓ python punto flotante
  - ✓ modelado ideal (usando conv2d) (usa frame completo)
  - ✓ modelado teórico (descomposición de la convolución) (usando fors)
  - ✓ si todo funciona probar con server (integración)
- ✓ python punto fijo del modelado teórico
- ✓ program logic verilog
  - ✓ describir la convolución y hacer verificación haciendo vector matching con simulación en python en punto fijo
- ✓ incorporar bram
- ✓ integrar módulos y crear un testbench de integracion
- ✓ actualizar simulaciones en python teniendo en cuenta la arquitectura
- ✓ Probar script con otros kernel
- ✓ Probar script con otros tamaños de matriz
- ✓ Crear módulo en base al python
- ✓ Integrar con interfaz axi stream
- ✓ admitir cambios de kernel desde el microblaze
- ✓ guardar proyecto usando archivo .tcl y hacer el control de version de hardware con git