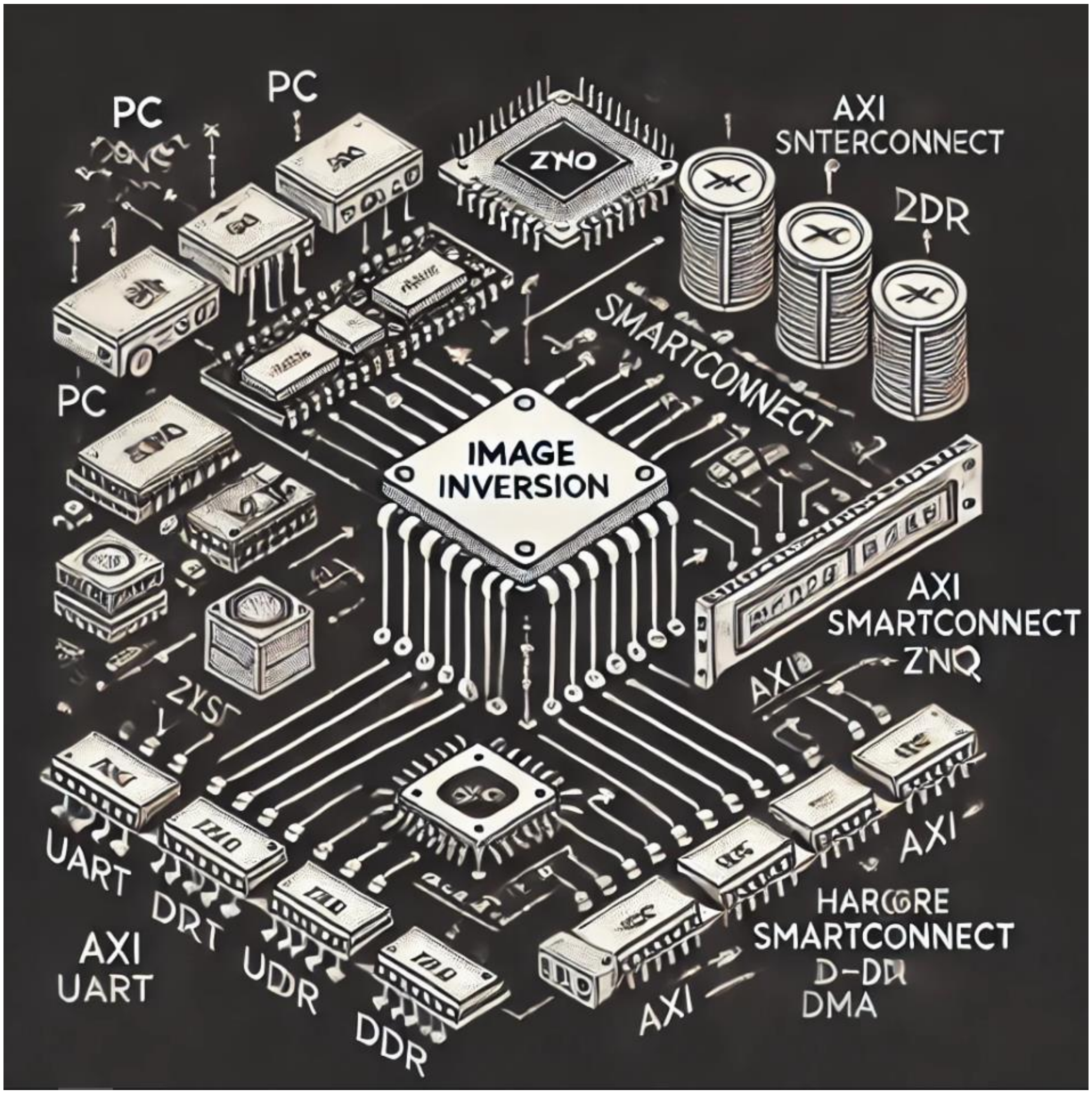
ZYNQ HW-SW Image Inversion



Created By:

Mohamed Ahmed Mohamed Hussein 11/2/2024

Introduction to Hardware-Software Co-Design

1. What is Hardware-Software Co-Design?

Hardware-Software Co-Design is a design methodology that integrates **hardware and software development** to optimize system performance, flexibility, and efficiency. It involves partitioning system functionalities between **software running on a processor (PS - Processing System)** and **hardware implemented on an FPGA (PL - Programmable Logic)**.

In this approach, **software handles control logic and configuration**, while **hardware accelerates computationally intensive tasks** such as **image processing, machine learning, and high-speed data handling**.

---

2. Why is Hardware-Software Co-Design Important?

- **Optimized Performance:** Offloads computationally expensive tasks to dedicated hardware, reducing execution time.
- **Efficient Resource Utilization:** Ensures a balance between general-purpose software flexibility and specialized hardware efficiency.
- **Parallel Processing:** Hardware modules (implemented in FPGA) can operate in parallel, significantly improving throughput.
- **Scalability:** Supports modular designs where new hardware blocks (IP cores) can be added without changing the overall system architecture.

---

3. Applications of Hardware-Software Co-Design

🔹 **Image and Video Processing** – Real-time filtering, edge detection, image compression.
🔹 **Embedded Systems** – Automotive ECUs, IoT devices, industrial

automation.
❖ **Networking and Telecommunications** – High-speed packet processing, error correction.
❖ **Artificial Intelligence and Machine Learning** – Hardware accelerators for deep learning.
❖ **Medical and Scientific Computing** – Signal processing for medical imaging, bioinformatics.

In this project, **hardware (PL) accelerates image inversion**, while **software (PS) configures and controls the operation** through an AXI-based communication interface.

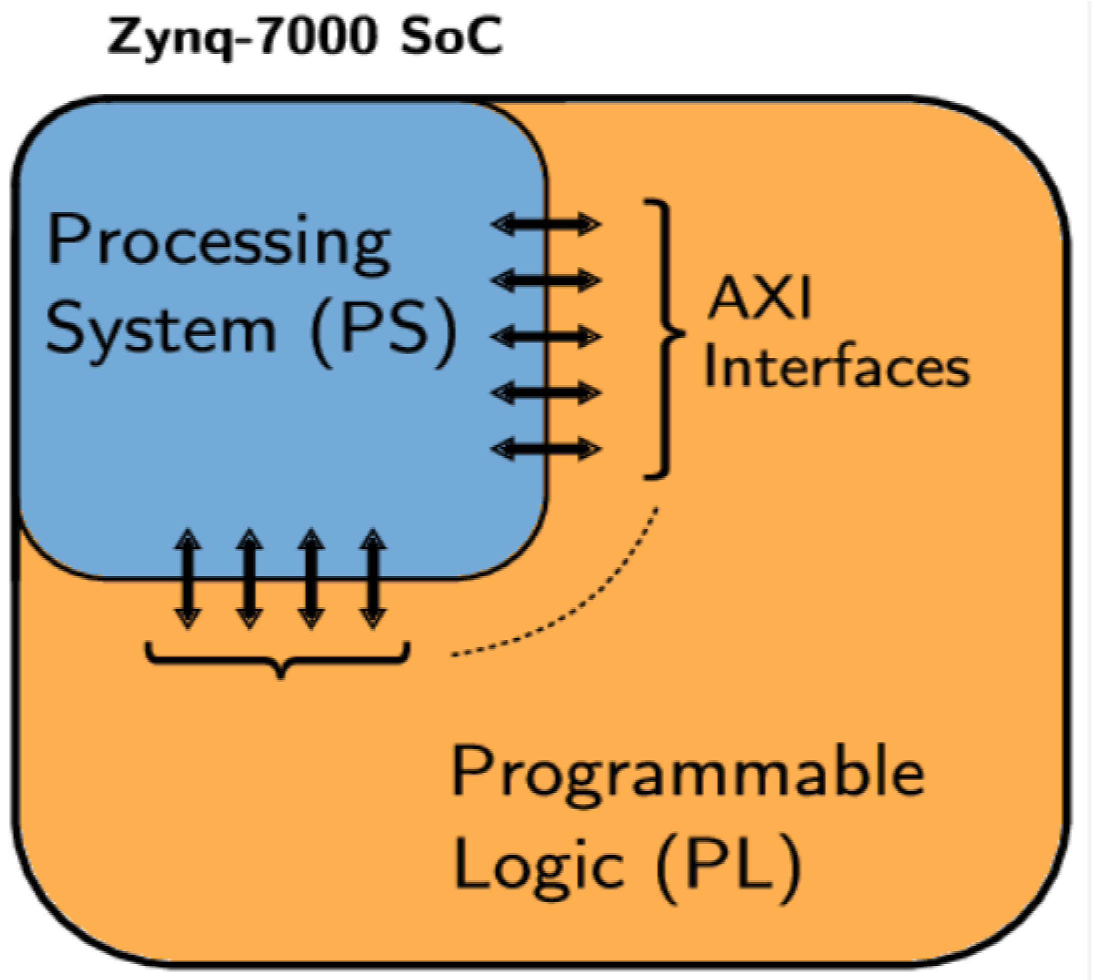
Idea of the Project

The main objective of this project is to process a given image by applying an **inversion operation**. Image inversion refers to the process of flipping pixel values such that bright areas become dark and vice versa. This transformation is commonly used in various image processing applications such as medical imaging, visual enhancements, and data analysis. In this project, we handle the image as a raw data file, where each pixel is represented as a single byte. The custom IP developed will facilitate the inversion operation efficiently, ensuring correct data transfer and processing.

Understanding Key Components

To fully grasp the **Image Inversion IP block**, it is essential to understand the following components:

1. PS-PL Configuration



The **Zynq Processing System (PS) and Programmable Logic (PL) communicate** via various AXI interfaces. The **CPU (PS) configures the PL hardware (FPGA)** and manages data transfer.

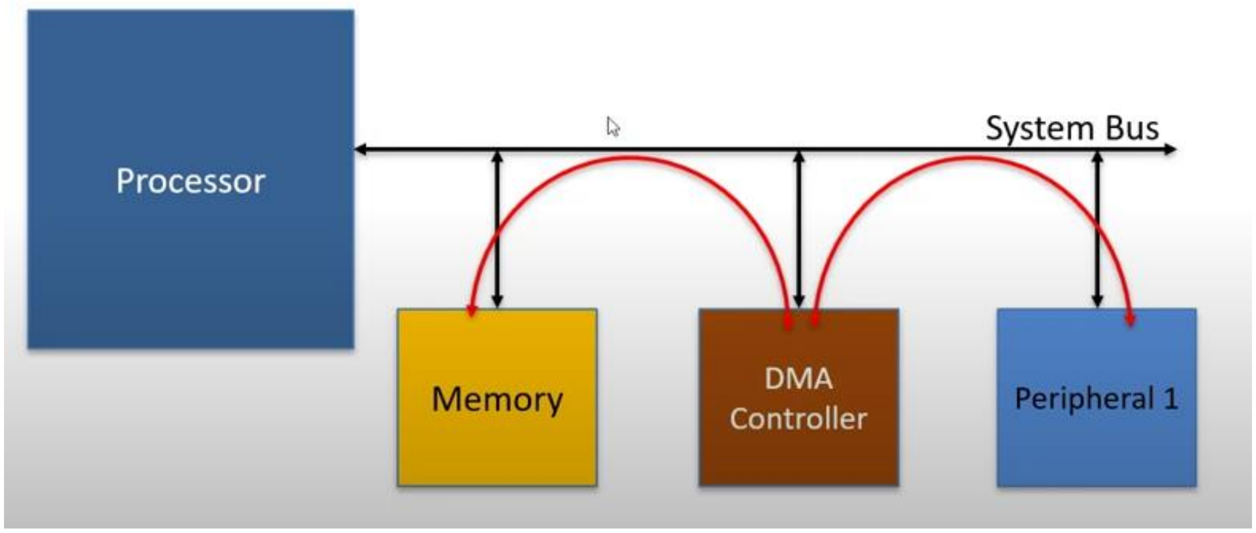
📌 General Function

The PS-PL configuration defines how software (PS) and hardware (PL) communicate **via** AXI interfaces (AXI-Lite, AXI4, AXI-Stream)**.**

📌 Function in the Project:

- The **CPU writes configuration settings** to AXI DMA using **AXI-Lite** (via AXI Interconnect).
- The **AXI DMA handles bulk data transfers** between **DDR (PS) and Custom IP (PL)**.
- Ensures **efficient software-hardware synchronization**.

2. AXI Direct Memory Access (AXI DMA)



📌 General Function:

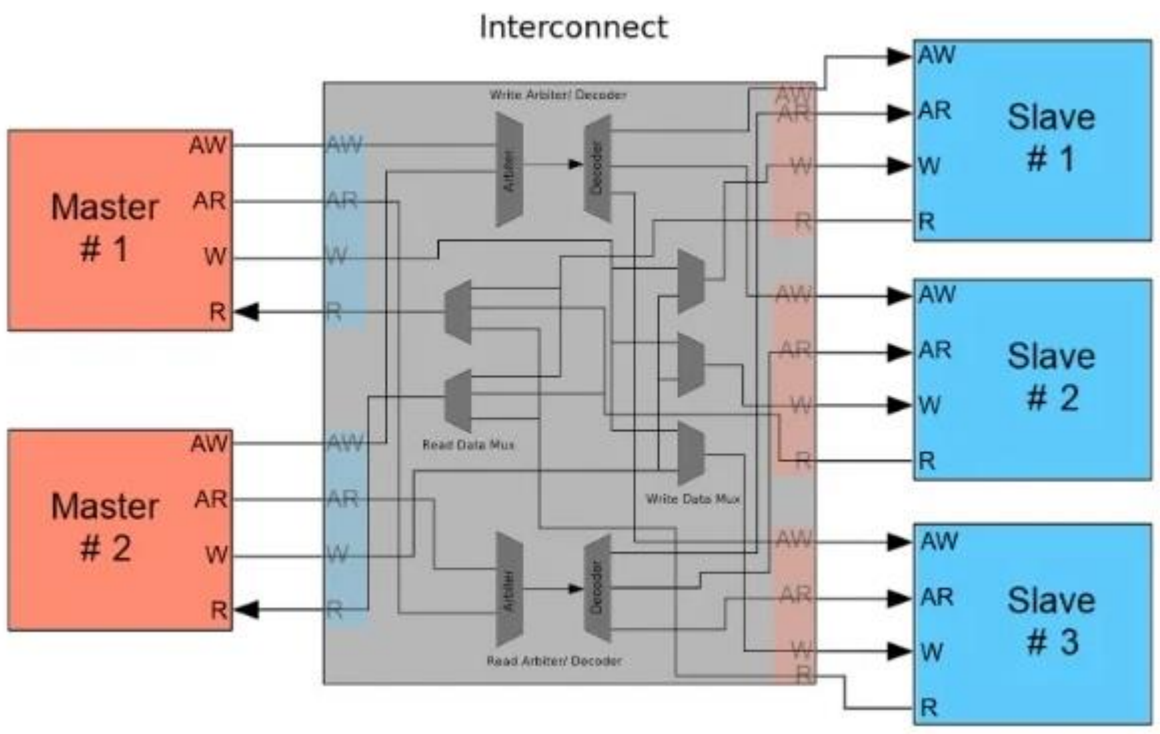
AXI DMA is a **direct memory transfer engine** that moves large amounts of data between **DDR memory and AXI-Stream peripherals** without CPU intervention. It significantly improves performance by allowing **direct data transfer**. It has two main channels:

- **MM2S (Memory-Mapped to Stream):** Reads from memory and sends to a peripheral.
- **S2MM (Stream to Memory-Mapped):** Receives data from a peripheral and writes to memory.

📌 Function in the Project:

- **Reads image data from DDR** and sends it to the **Custom IP (HW_SW_IMAGE_INVERSION)** via **AXI4-Stream**.
- **Receives processed data from the Custom IP** and writes it **back to DDR**.
- Configured by the CPU via **AXI-Lite**.

3. AXI Interconnect



AXI Interconnect acts as a **bridge between the Processing System (PS) and the FPGA (PL).** It allows the CPU (PS) to access various peripherals and memory-mapped IP cores within the FPGA.

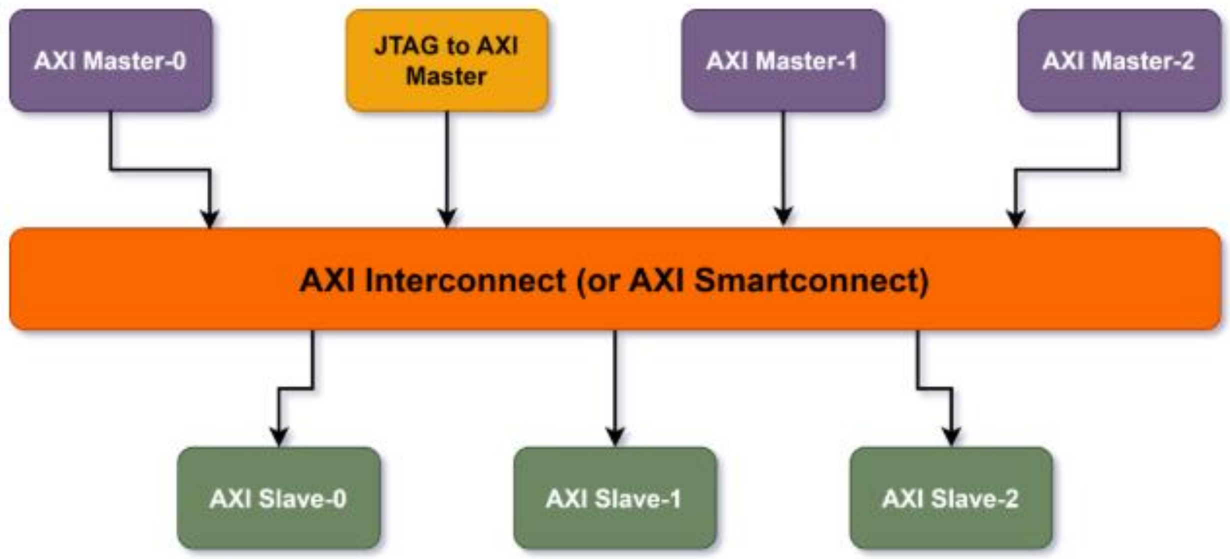
📌 General Function:

AXI Interconnect is a **bus matrix** that enables multiple **AXI masters (e.g., CPU, DMA) to communicate with multiple AXI slaves (e.g., memory, peripherals).** It supports **address decoding, data routing, and arbitration** between multiple transactions.

📌 Function in the Project:

- Allows the CPU (PS) to configure AXI DMA **via** AXI-Lite**.**
- **ables seamless communication between** PS and PL**.**
- **Ensures correct** memory mapping of the Custom Image Inversion IP**.**

4. AXI SmartConnect



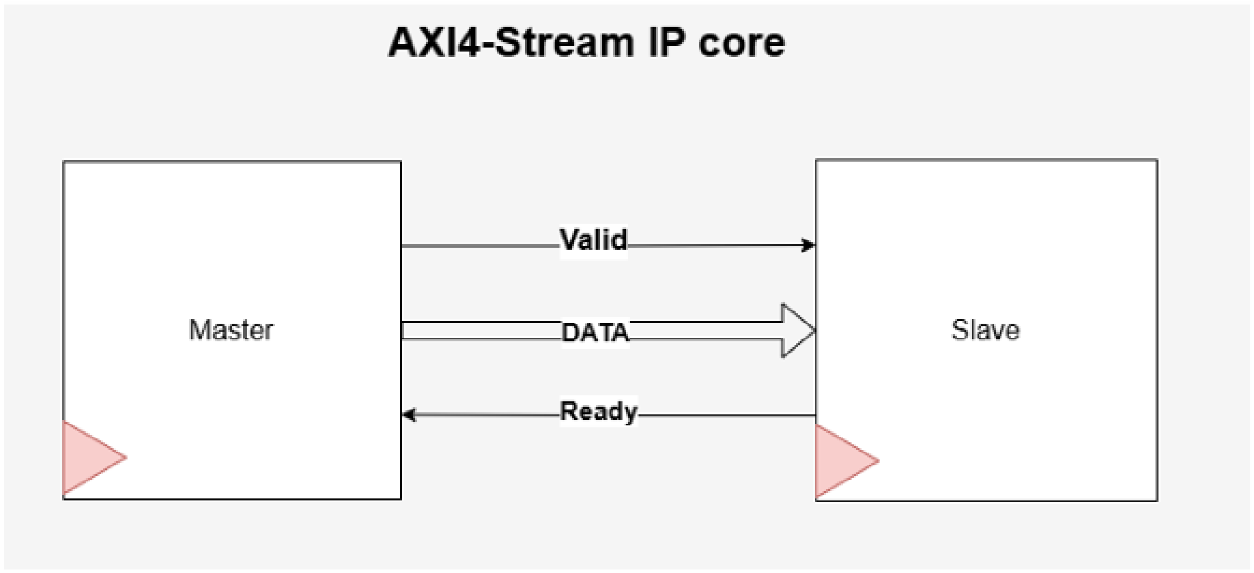
AXI SmartConnect is an **interconnect module** that dynamically manages **multiple AXI interfaces** with **high efficiency**. It is used to connect **AXI Masters (such as DMA) to AXI Slaves (such as DDR memory or custom IPs)**.

📌 Function in the Project:

- **Routes** data transfer requests **between the** AXI DMA **and** DDR Memory**.**

- **Supports** high-performance burst transfers **for optimized data movement.**
- **Reduces** bus contention **by efficiently handling multiple transactions.**

5. AXI4-Stream



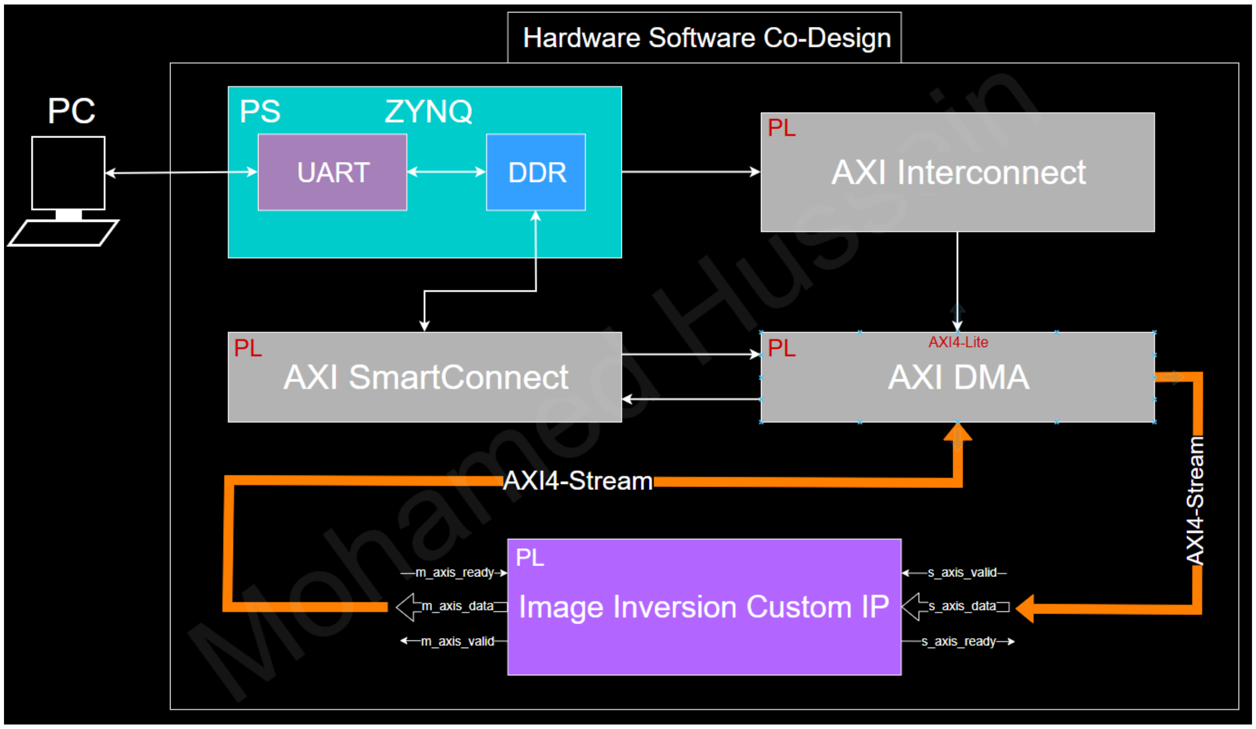
📌 General Function:

AXI-Stream is a **data streaming interface** optimized for **continuous, high-speed data flow**. Unlike AXI Memory-Mapped, AXI-Stream does not use addresses; instead, it allows **packet-based communication**, making it ideal for processing pipelines.

📌 Function in the Project:

- **Transports image data** between **AXI DMA** and **the Custom Image Inversion IP**.
- Enables **real-time pixel streaming** for efficient processing.
- Ensures minimal latency compared to traditional memory-mapped transfers.

Diagram and Explanation



This diagram represents the **Hardware-Software Co-Design** implementation of an **Image Inversion IP** using a **Zynq-based system**. The process begins when an image is sent from the **PC** and follows multiple steps through both the **Processing System (PS)** and **Programmable Logic (PL)** before the inverted image is returned to the PC.

Step-by-Step Data Flow Explanation:

1. PC Sends the Image to the Zynq Processing System (PS)

- The **PC transmits the image** via **UART**.
- The **C code running on the Zynq processor** receives the image data and stores it in **DDR Memory** inside the PS.

2. Zynq Processor Configures the AXI DMA via AXI4-Lite

- The **C code configures the AXI DMA** through **AXI4-Lite**.
- The configuration is performed via **AXI Interconnect**, allowing the CPU to set: **Source Address** (DDR Memory where the image is stored).

Destination Address (DDR location for the processed image).
Transfer Size (amount of image data).
Start Command (initiates DMA transfer).

3. AXI DMA Reads the Image from DDR Memory via AXI SmartConnect

- After being configured, the **AXI DMA fetches the image from DDR Memory** for processing.
- The **AXI SmartConnect module ensures the data is properly routed** from memory to the next stage.
- The data is formatted into a **continuous stream** so that the FPGA can process it efficiently.

4. Image Data is Transferred to the Image Inversion Custom IP via AXI4-Stream

- The **AXI4-Stream protocol** is used to send pixel data to the **Image Inversion Custom IP**.
- The **Custom IP receives the image data** through:
  - s_axis_valid → Indicates valid incoming data.
  - s_axis_data → Contains the pixel values.
  - s_axis_ready → Synchronizes data flow.

5. Image Processing in the Custom IP (Inversion Logic in PL)

- The **Image Inversion Custom IP processes the incoming data**, applying **pixel inversion (pixel_out = 255 - pixel_in)**.
- The processed (inverted) image is **sent back via AXI4-Stream** using:
  - m_axis_valid → Indicates valid output data.
  - m_axis_data → Contains the inverted pixel values.
  - m_axis_ready → Ensures synchronization.

6. AXI DMA Receives the Inverted Image and Writes it Back to DDR via AXI SmartConnect

- The **inverted image is sent back from the Custom IP using AXI4-Stream**.
- The **AXI DMA receives the processed data and writes it back into DDR Memory**.
- The **AXI SmartConnect module ensures the data is properly routed to memory** for storage.

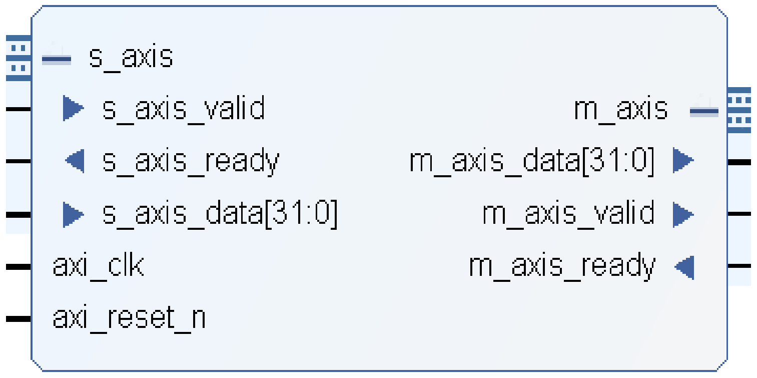
7. The Zynq Processor Sends the Inverted Image Back to the PC

- The **Zynq Processor retrieves the processed image from DDR Memory**.
- The image is then **sent back to the PC via UART communication**.
- The **PC receives the inverted image, completing the processing cycle**.

Key Takeaways from the Diagram:

- The **PC sends the image to the Zynq Processing System (PS) via UART**.
- The **Processing System (PS) controls data movement and configures AXI DMA**.
- The AXI SmartConnect **ensures smooth communication between memory, AXI DMA, and the Custom IP**.
- The Image Inversion Custom IP **processes the image inside the FPGA (PL)**.
- The AXI DMA **moves data efficiently between DDR Memory and the Custom IP using AXI4-Stream**.
- The **processed image is sent back to the PC via UART, completing the cycle**.

Our IP



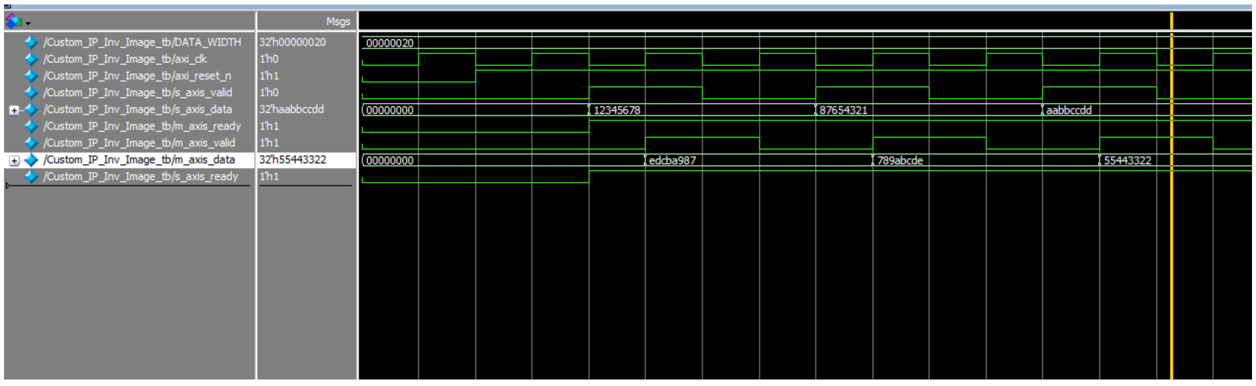
Function:

Our custom IP is designed to perform **image inversion** at the hardware level efficiently. It processes image data pixel by pixel, modifying each pixel's value to achieve the desired inversion effect.

Explanation of Image Inversion Process:

1. **Pixel Representation:**
   - Each **pixel** in the image is represented as **1 byte (8 bits)**.
   - This byte typically corresponds to a grayscale intensity value (0–255), where **0** represents black and **255** represents white.
2. **Inversion Logic:**
   - The core function of the IP is to **invert** the pixel value by applying the formula: Inverted Pixel=255−Original Pixel\text{Inverted Pixel} = 255 - \text{Original Pixel}Inverted Pixel=255−Original Pixel
   - If a pixel is **completely black (0)**, after inversion, it becomes **completely white (255)** and vice versa.
   - Intermediate shades are also flipped accordingly.
3. **Data Flow:**
   - The image data is **received** from memory (e.g., DDR or BRAM).
   - Each pixel is **processed** through the custom IP, where the inversion occurs in hardware.
   - The inverted pixel values are **stored back** in memory for further use or display.
4. **Why Hardware Acceleration?**
   - Performing image inversion in hardware significantly improves speed compared to software execution.
   - The IP operates in a **pipelined** or **parallelized** manner, allowing multiple pixels to be processed simultaneously, making it highly efficient.

Test Bench:

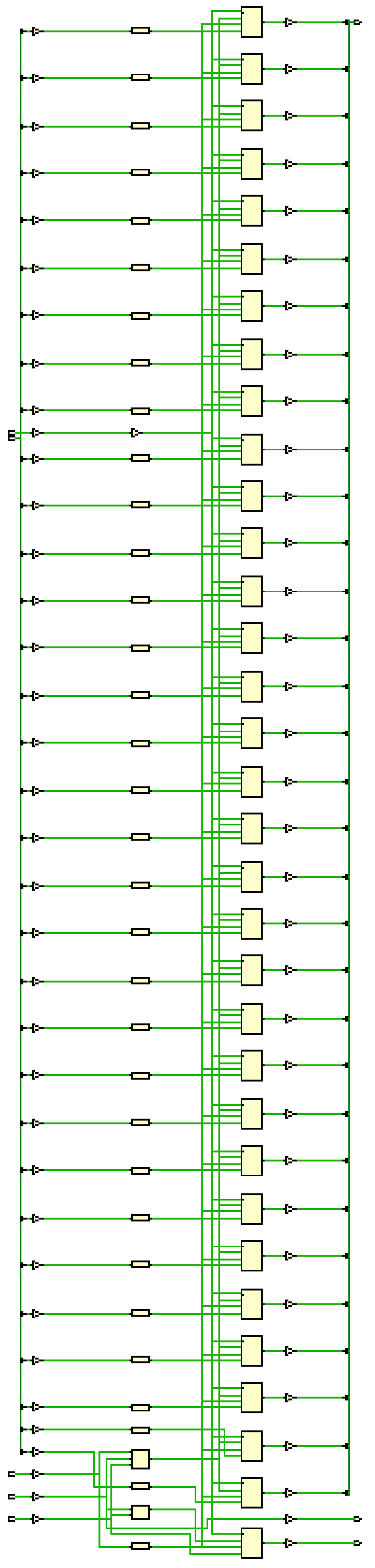


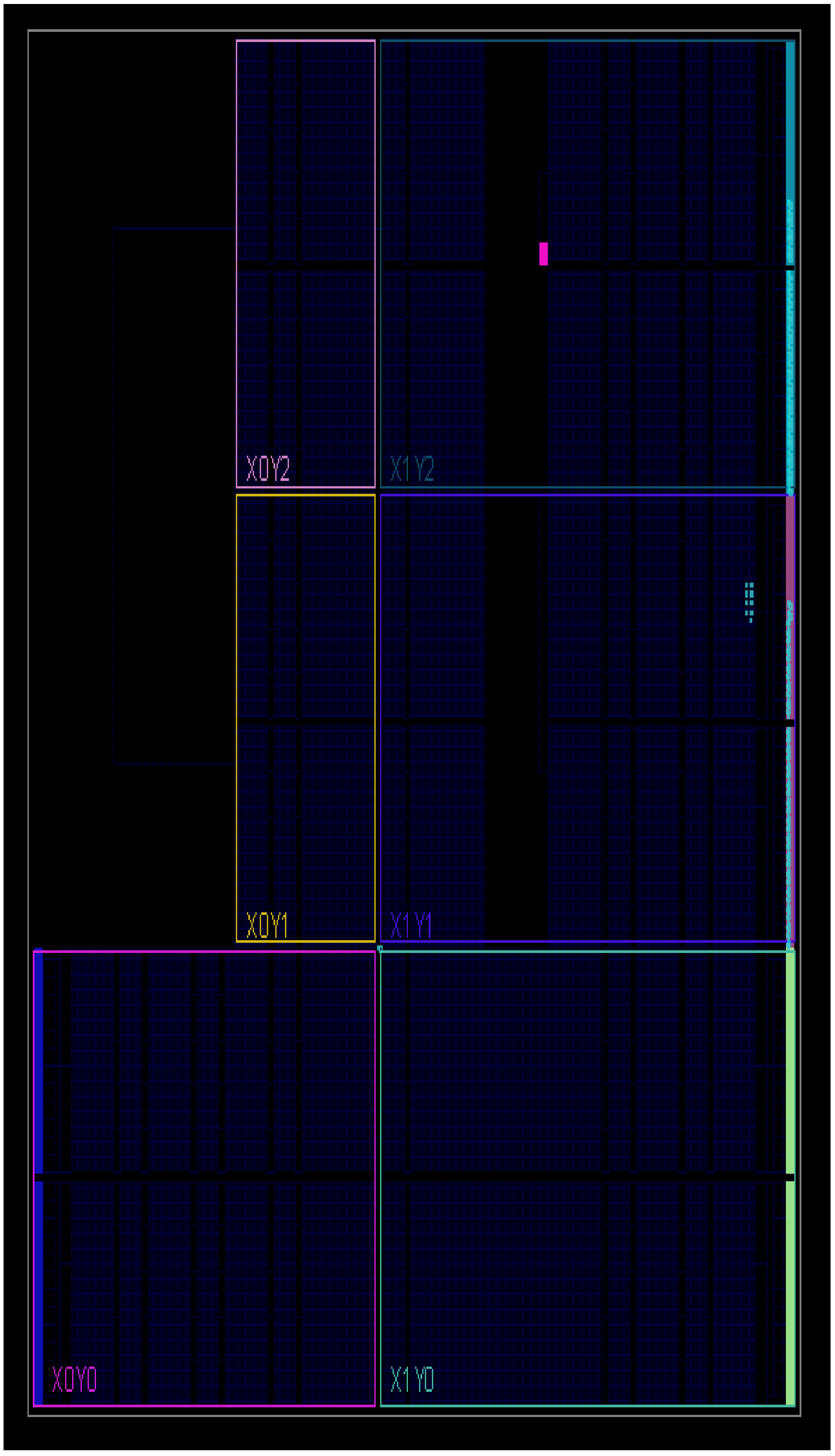
```
                                              Msgs
/Custom_IP_Inv_Image_tb/DATA_WIDTH    32'h00000020   00000020
/Custom_IP_Inv_Image_tb/axi_clk       1'h0
/Custom_IP_Inv_Image_tb/axi_reset_n   1'h1
/Custom_IP_Inv_Image_tb/s_axis_valid  1'h0
/Custom_IP_Inv_Image_tb/s_axis_data   32'haabbccdd   00000000  12345678  87654321  aabbccdd
/Custom_IP_Inv_Image_tb/m_axis_ready  1'h1
/Custom_IP_Inv_Image_tb/m_axis_valid  1'h1
/Custom_IP_Inv_Image_tb/m_axis_data   32'h55443322   00000000  edcba987  789abcde  55443322
/Custom_IP_Inv_Image_tb/s_axis_ready  1'h1
```

We will see that every Byte is inverted

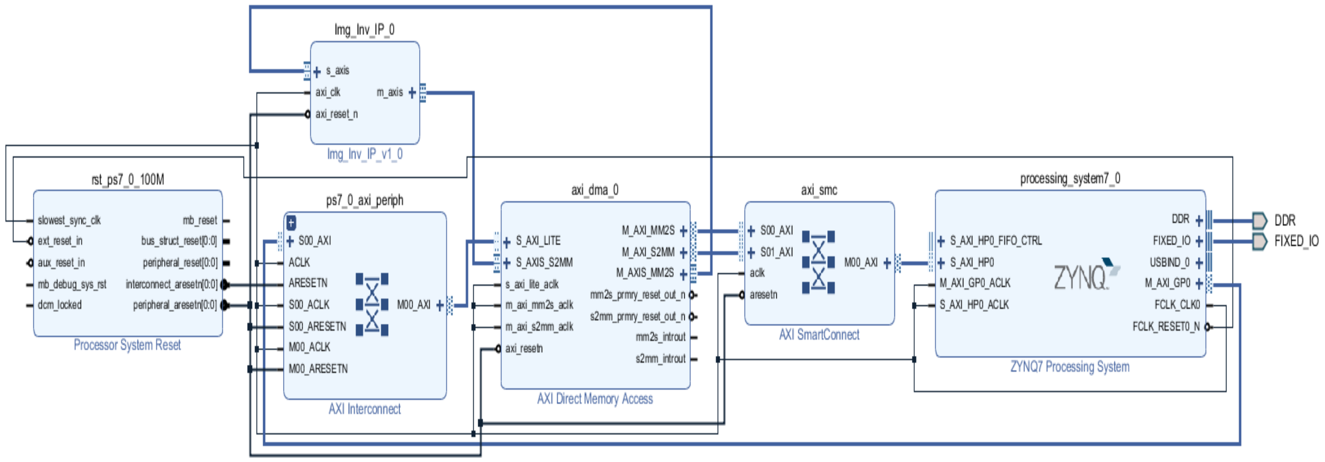
Synthesis:

Implementation:

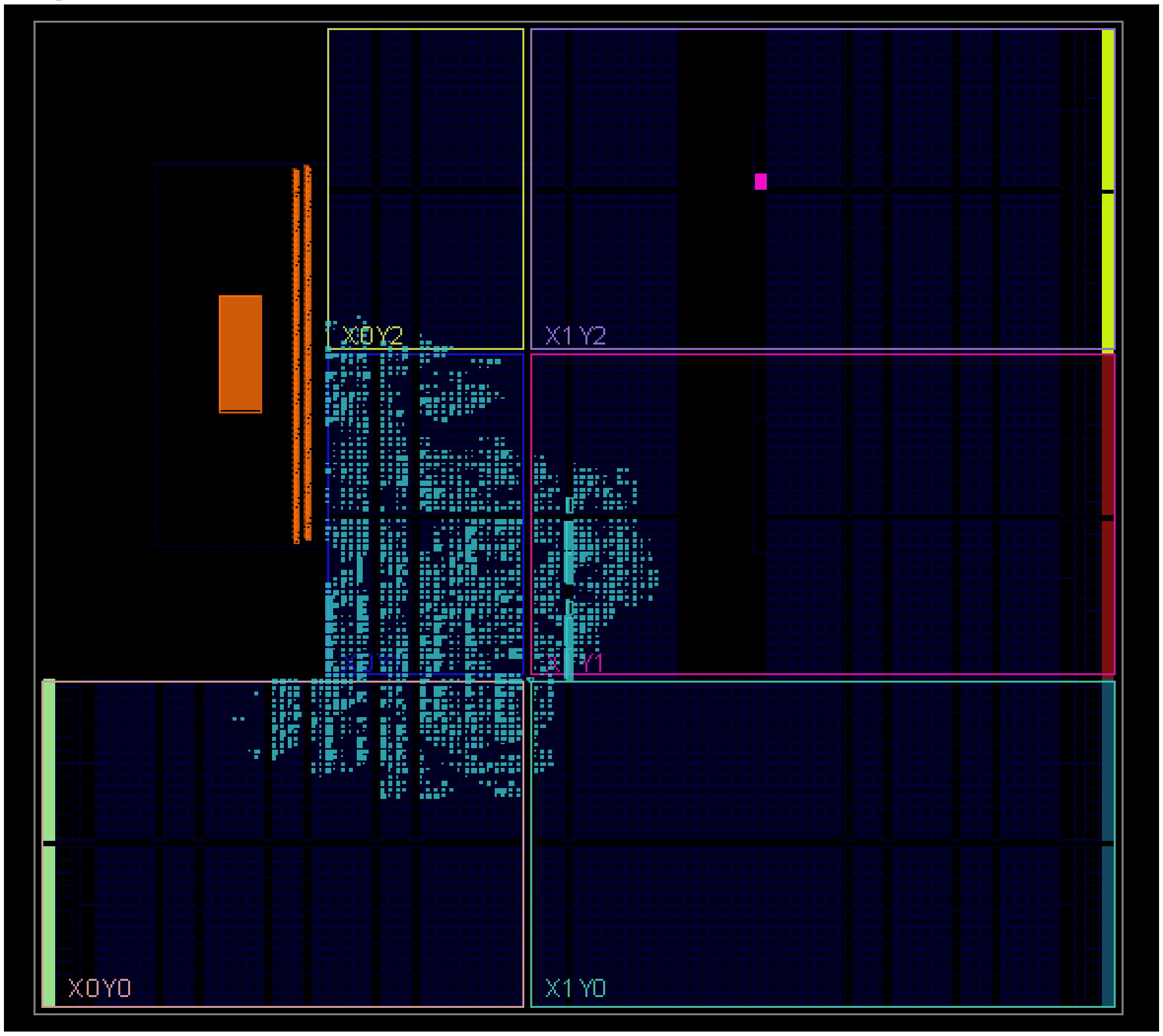




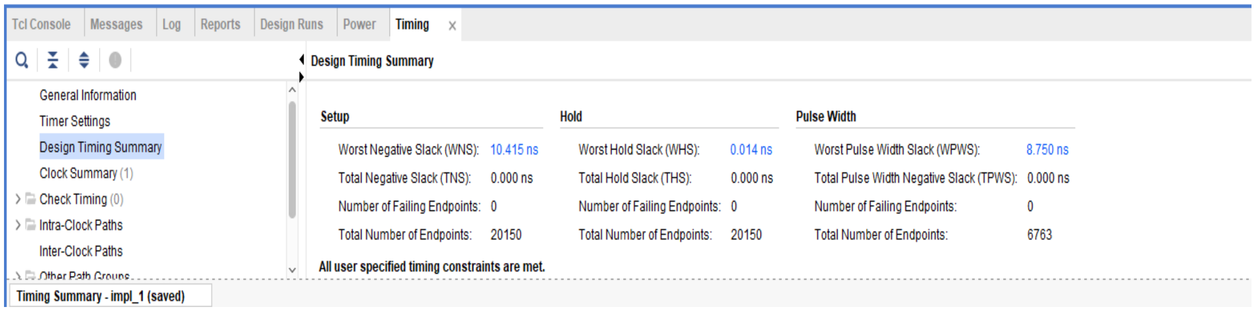
Full Design



Implementation:



Timing:

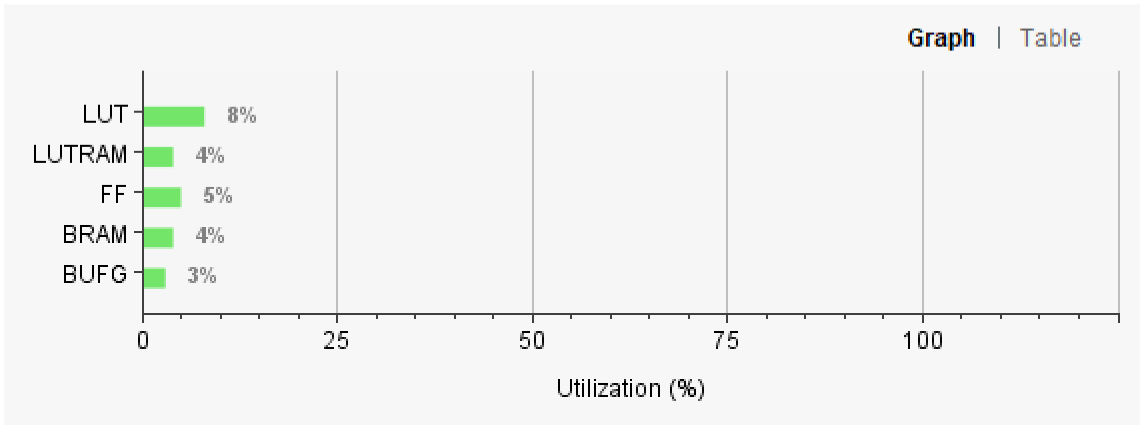


Design Timing Summary

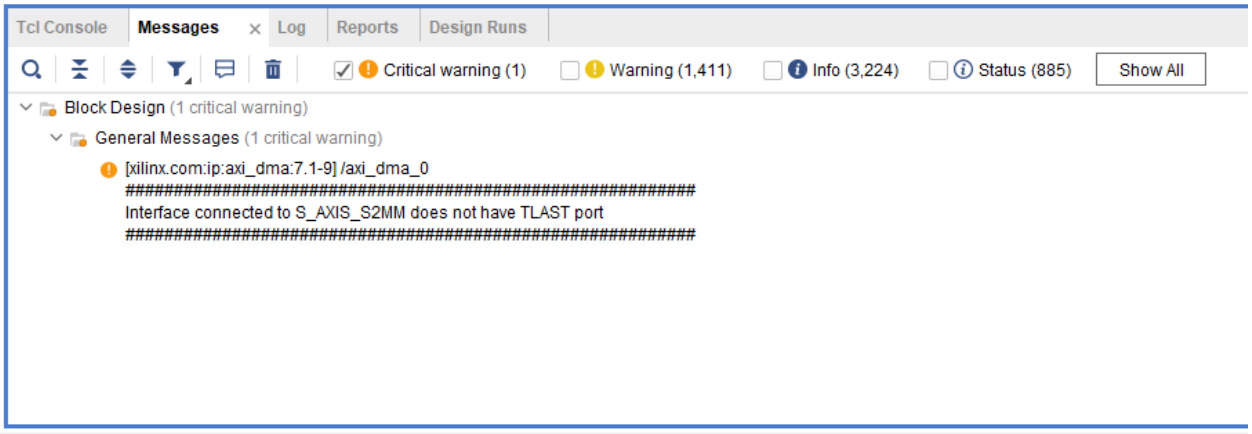
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	10.415 ns	Worst Hold Slack (WHS):	0.014 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	20150	Total Number of Endpoints:	20150	Total Number of Endpoints:	6763

All user specified timing constraints are met.

Utilization:



Messages:



```
Block Design (1 critical warning)
  General Messages (1 critical warning)
    [xilinx.com:ip:axi_dma:7.1-9] /axi_dma_0
    ##################################################################
    Interface connected to S_AXIS_S2MM does not have TLAST port
    ##################################################################
```

The **critical warning** about TLAST is acceptable because your Image Inversion IP processes **fixed-size images** (e.g., 512×512 pixels, 1 byte per pixel). Since the data size is predefined, AXI DMA can transfer the full image without needing TLAST, operating in **non-packet mode**. This warning **is safe to ignore** as long as DMA is configured for fixed-length transfers

SDK (Software Development Kit)

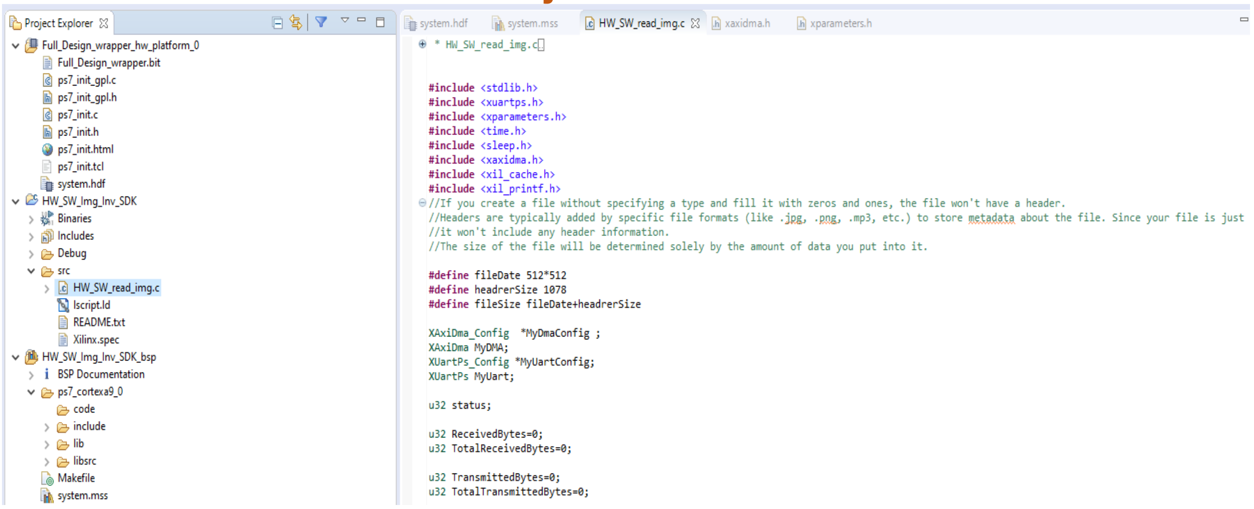
Introduction:

A **Software Development Kit (SDK)** is a collection of tools, libraries, and documentation that facilitates the development and interaction with hardware or software components. It plays a crucial role in embedded systems and FPGA-based projects, ensuring smooth communication between software and hardware.

General Importance of SDK in Embedded Systems:

1. **Enables Software-Hardware Interaction:**
   - SDK provides APIs and drivers that allow software to configure and control custom hardware components.
2. **Speeds Up Development:**
   - Instead of writing low-level code from scratch, developers use SDK functions to streamline software implementation.
3. **Provides Debugging Tools:**
   - SDKs include debugging utilities that help identify and fix issues in the communication between hardware and software.
4. **Standardized Interfaces:**
   - SDKs ensure consistency in software design, allowing seamless integration with different hardware peripherals.

Function of SDK in the Project:



```c
* HW_SW_read_img.c


#include <stdlib.h>
#include <xuartps.h>
#include <xparameters.h>
#include <time.h>
#include <sleep.h>
#include <xaxidma.h>
#include <xil_cache.h>
#include <xil_printf.h>
//If you create a file without specifying a type and fill it with zeros and ones, the file won't have a header.
//Headers are typically added by specific file formats (like .jpg, .png, .mp3, etc.) to store metadata about the file. Since your file is just
//it won't include any header information.
//The size of the file will be determined solely by the amount of data you put into it.

#define fileDate 512*512
#define headrerSize 1078
#define fileSize fileDate+headrerSize

XAxiDma_Config  *MyDmaConfig ;
XAxiDma MyDMA;
XUartPs_Config *MyUartConfig;
XUartPs MyUart;

u32 status;

u32 ReceivedBytes=0;
u32 TotalReceivedBytes=0;

u32 TransmittedBytes=0;
u32 TotalTransmittedBytes=0;
```

The **Software Development Kit (SDK)** plays a crucial role in the Image Inversion project by enabling the execution and management of the C code. It provides the necessary tools and environment for software-hardware interaction, ensuring efficient operation of the designed system. Below are the key functions of SDK in this project:

1. **Hardware Integration**
    - The SDK includes the **hardware description files (HDF)** generated from Vivado.
    - These files allow the software to properly interact with the custom **Image Inversion IP**, ensuring correct data processing and hardware communication.
2. **Driver & BSP Support**
    - The SDK provides **Board Support Packages (BSPs)** that include essential **drivers** for peripherals such as **AXI DMA and UART**.
    - These drivers facilitate smooth data transfer between the FPGA hardware and the processing system.
3. **Code Compilation & Debugging**
    - The SDK enables writing, compiling, and debugging of the **HW_SW_read_img.c** file.
    - It ensures proper **DMA operations** for reading and processing image data efficiently.
4. **Application Deployment**
    - SDK allows seamless execution of the software application on the FPGA-based system.
    - It provides debugging tools to ensure the program runs as expected, facilitating quick issue resolution.

In summary, the **SDK acts as a bridge between the hardware (FPGA) and software (C program)**, enabling the **Image Inversion project** to function effectively.

Function of C_Code (extra)

1. **Allocates Memory for the Image**
    - The image data (fileData) is stored in memory, including the **header (1078 bytes) and pixel data (512x512 bytes).**

2. **Initializes AXI DMA and UART**
    - **AXI DMA** is set up for high-speed data transfer between **DDR memory and the FPGA PL.**
    - **UART** is configured for serial communication with the PC (baud rate: **115200**).

3. **Receives Image Data from the PC via UART**
    - The **PC sends the image** to the Zynq system through UART.
    - The **Zynq processor stores the received data** in the allocated memory (fileData).
    - This process continues until the entire image is received.

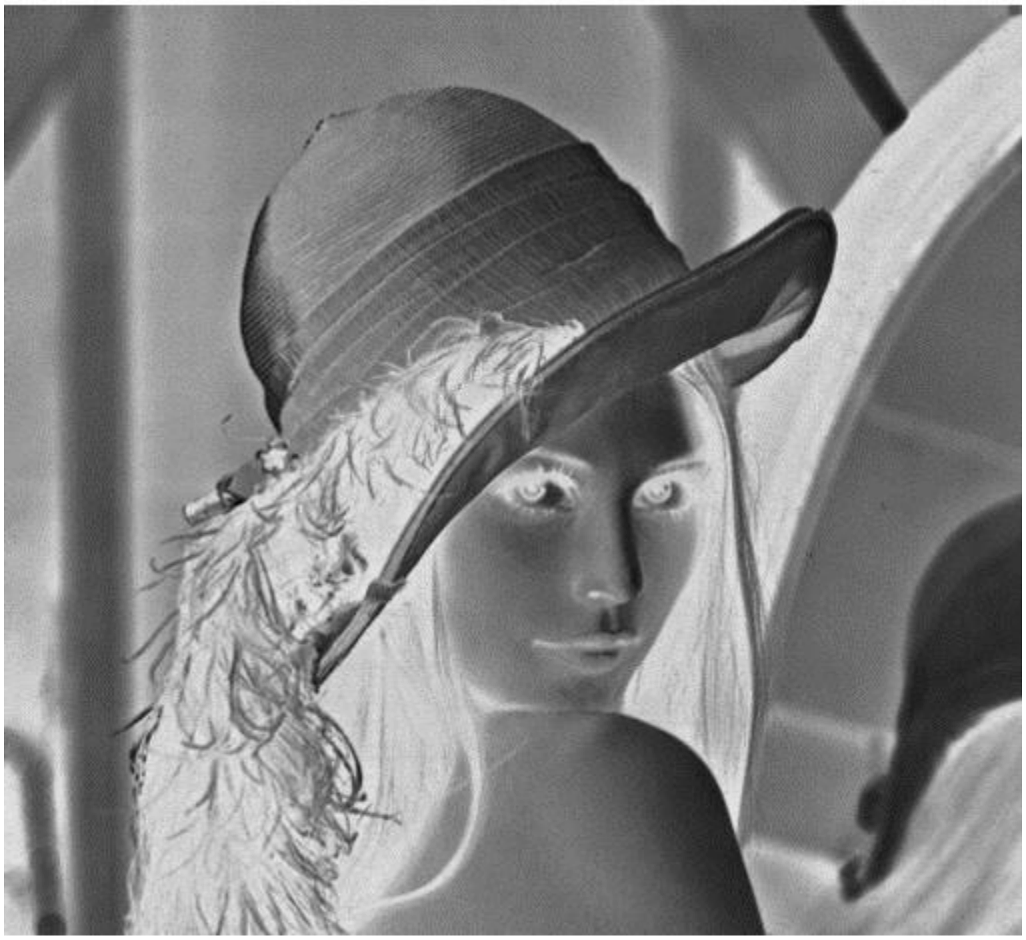
4. **Flushes Cache to Ensure Data is Written to Memory**
   - Ensures that the processor writes fileData **immediately to memory** before AXI DMA reads it.

5. **Sends Image Data to the FPGA via AXI DMA**
   - **AXI DMA is configured to transfer image data** (excluding the header) from DDR memory to **the Image Processing IP.**
   - The DMA is set to transfer data in **both directions:**
     - **Device-to-DMA (Sending Image to FPGA).**
     - **DMA-to-Device (Receiving Processed Image from FPGA).**

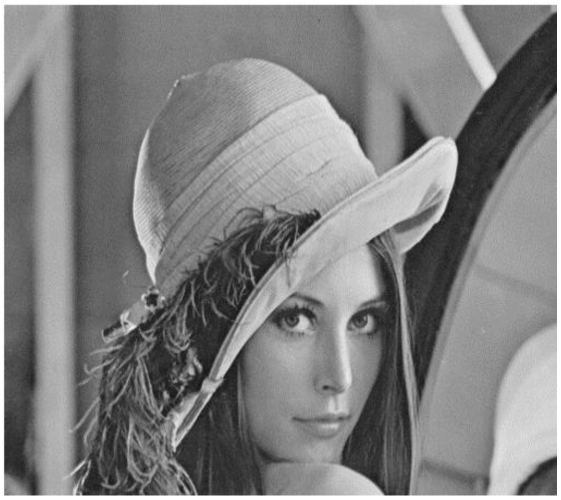
6. **Waits for the DMA Transfer to Complete**
   - The code **checks if DMA has halted**, ensuring that the data is fully transferred before proceeding.

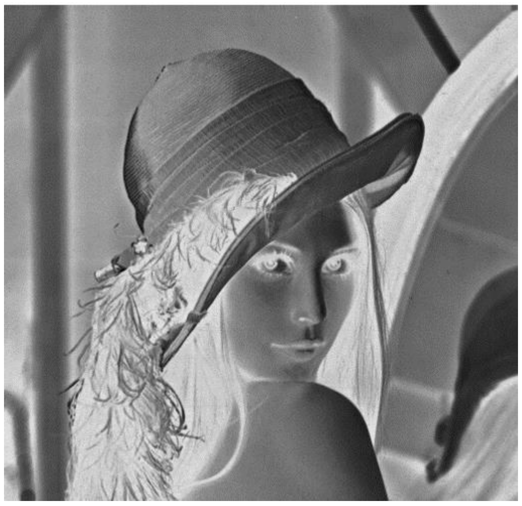
7. **Sends the Processed Image Back to the PC via UART**
   - The **processed image is sent back** to the PC, 2 bytes at a time.
   - **A small delay (usleep(1000)) is added** to avoid overflow in UART transmission.

Output



Compare





Inversion Process done Correctly

Conclusion

In this project, we successfully designed and implemented a custom hardware IP for image inversion, leveraging FPGA resources and the AXI DMA interface for efficient data transfer. The project demonstrated the ability to process images pixel-by-pixel, where each pixel is treated as a single byte, ensuring seamless inversion.

The integration of the hardware design with the Software Development Kit (SDK) enabled the execution of C-based control logic, managing data flow between the processing system and programmable logic. Through this approach, we validated the functionality of our custom IP, ensuring correct image inversion.

This work highlights the importance of hardware-software co-design in embedded systems and FPGA-based applications, demonstrating the effectiveness of custom IPs for specialized image processing tasks.