MTP-1 Project Report

Hardware Accelerated Network Traffic Analytics

Department of Computer Science and Engineering

IIT Madras

Submitted by

Baddala Sai Kumar Reddy
CS23M059

Parthasaradhi
CS23M035

Guided by

Prof. Ayon Chakraborty

November 2024

Abstract

In the era of rapidly expanding network traffic, the demand for real-time analytics has become critical to maintain network performance, security, and user experience. This project aims to harness the computational power of Deep Neural Networks (DNNs) and Field Programmable Gate Arrays (FPGAs) to create a high-speed, efficient **Network Packet Analysis** system. The goal is to design and implement a module capable of analyzing network packets at line rate by leveraging DNNs fine-tuned for network traffic analysis tasks. These tasks include the detection of **anomalous traffic patterns**, **classification of traffic types** (such as video streaming versus web browsing), and **prediction of Quality of Experience** (QoE) metrics.

Central to this project is the use of the **Xilinx Zynq 7000 SoC**, which integrates a dual-core ARM Cortex-A9 processor with FPGA-based programmable logic.This **hardware-software co-design**allows the system to take advantage of the parallel processing capabilities and reconfigurability of FPGAs to accelerate DNN inference and network packet processing tasks. By offloading computationally intensive operations to the FPGA, the design aims to achieve significant speedups and reduce latency, thereby making real-time analytics feasible in high-throughput network environments.

The primary objective of this M.Tech project is to develop an **embedded Linux-based platform** that functions as an **AI-accelerated Network Interface Card (NIC)**. This platform will perform real-time analytics on network traffic, enabling efficient monitoring and management. The FPGA-accelerated NIC will not only improve the responsiveness of network analysis but also provide a scalable solution adaptable to evolving network demands. Through this approach, the project demonstrates the potential of FPGA-based acceleration for enhancing performance in machine learning-driven network analytics.

Contents

Chapter 1

Problem Statement

1.1. Problem Statement

1.1.1. Objective

The objective of this project is to develop a solution for real-time network traffic analytics that can handle tasks such as **packet identification, traffic classification, and anomaly detection**. These analytics are critical for enhancing network security and optimizing performance by providing insights into network behavior and identifying potential threats or inefficiencies. With the growth in network traffic and the increased complexity of modern cyber threats, real-time analytics have become essential for maintaining robust and secure networks.

1.1.2. Challenges with Traditional Routers

While routers are a central component of network infrastructure, they are primarily designed to forward packets efficiently and are not well-suited for advanced analytics. Attempting to use routers for real-time analytics presents several challenges:

- **Limited Analytics Capability:** Traditional routers are designed for packet routing and switching, not for performing in-depth analytics or AI-based processing. This limitation means routers lack the ability to detect patterns, classify traffic, or identify threats effectively, making them insufficient for complex network monitoring tasks.
- **High Latency:** When routers are tasked with additional analytics functions, their latency increases due to the lack of specialized processing units. Routers do not have dedicated hardware for high-speed data analysis, which results in slower performance and delays in identifying critical events, impacting real-time response capabilities.
- **Processing Constraints:** Running complex AI models on traditional routers is impractical as they do not have the necessary computational power. Routers typically rely on general-purpose CPUs with limited resources, which are unable to handle the demands of machine learning or deep learning algorithms required for accurate network analysis.

- **Scalability Issues:** As network speeds increase and data volumes grow, routers struggle to keep pace. High-speed networks produce large volumes of data that require rapid processing, which traditional routers are not equipped to handle when tasked with additional analytics. This lack of scalability limits the effectiveness of routers in environments with high traffic loads or complex data analysis requirements.

1.1.3. Proposed Solution

To overcome these challenges, this project proposes the development of an AI-Accelerated Network Interface Card (NIC). This solution involves creating an embedded Linux platform that functions as an AI-driven NIC capable of handling network traffic analytics in real time. Here's how the AI-accelerated NIC addresses the limitations of traditional routers:

- **Dedicated Hardware for Analytics:** By integrating an FPGA on the NIC, we can offload computationally intensive tasks from the router to the NIC itself. FPGAs offer parallel processing capabilities that are ideal for running AI models efficiently, enabling tasks like packet identification, classification, and anomaly detection without impacting the router's core functions.
- **Reduced Latency:** The AI-accelerated NIC performs analytics directly on incoming packets as they pass through the NIC, reducing the need to send data to an external processor or the router for analysis. This reduces latency and allows for real-time response to potential threats or unusual traffic patterns.
- **Scalable and Efficient Processing:** The NIC is optimized for high-speed data transfer and is capable of handling large volumes of traffic due to its specialized hardware. With FPGA-based acceleration, it can scale to meet the demands of high-speed networks, enabling continuous analytics without bottlenecks.
- **AI-Driven Insights:** The AI-accelerated NIC can run lightweight machine learning models to analyze patterns, detect anomalies, and classify traffic types, providing AI-driven insights into network traffic in real time. This enables proactive threat detection, traffic optimization, and quality of service (QoS) adjustments, enhancing both network security and performance.

Chapter 2

Introduction

2.1. What is an FPGA?

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to be configured by the user after manufacturing. Unlike traditional Application-Specific Integrated Circuits (ASICs) that are hardwired for specific tasks, FPGAs offer reconfigurability, allowing developers to program their logic design to perform a range of complex operations. This flexibility is achieved through a network of programmable logic blocks and interconnects that can be configured to create custom hardware circuits.

FPGAs are particularly valuable in applications requiring parallel processing, low latency, and high customizability. They are commonly used in areas such as signal processing, data centers, machine learning, and embedded systems, where they can be tailored to accelerate specific computational tasks.

2.2. Overview of the PYNQ-Z2 Board

The PYNQ-Z2 board is a development platform that leverages the capabilities of the Xilinx Zynq-7000 SoC (System on Chip). The Zynq-7000 integrates a dual-core ARM Cortex-A9 processor (Processing System or PS) with FPGA-based programmable logic (Programmable Logic or PL). This combination allows for seamless hardware-software co-design, enabling developers to utilize both the PS for high-level software tasks and the PL for custom hardware acceleration.

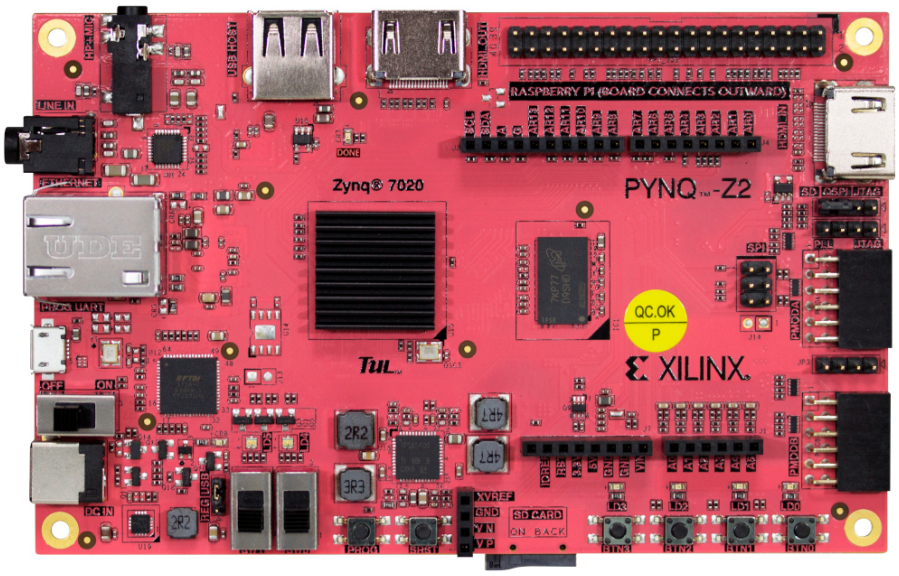

Figure 2.1: PYNQ-Z2

The PYNQ-Z2 board is designed to be user-friendly, featuring a range of peripherals such as HDMI, audio, and Ethernet ports. It also supports the PYNQ framework, which simplifies FPGA development by allowing users to interact with the board using Python scripts and Jupyter notebooks. This makes the PYNQ-Z2 an excellent platform for prototyping and educational purposes, where quick iteration and ease of use are essential.

2.3. Why FPGAs are Needed

Traditional CPUs are optimized for sequential processing, where tasks are executed one after another. While modern CPUs have multiple cores and can perform multithreading, their architecture is still fundamentally limited when it comes to true parallelism. This poses a challenge for applications that require massive parallel computations and real-time processing, such as deep learning inference, high-speed signal processing, and network traffic analysis.

FPGAs, on the other hand, excel in parallel processing. They allow for custom data paths and can execute multiple operations simultaneously, providing significant performance improvements for parallelizable tasks. Additionally, FPGAs offer low-latency responses, as their custom hardware can process data in real-time without the overhead associated with CPU instruction cycles.

2.4. Improving Performance with FPGAs

By leveraging the reconfigurability of FPGAs, developers can create hardware accelerators tailored to specific applications. This results in optimized performance and energy efficiency compared to using general-purpose CPUs. FPGAs can be programmed to handle data-intensive tasks directly in hardware, bypassing the

limitations of traditional instruction sets and enabling higher throughput and lower power consumption.

For example, in machine learning applications, FPGAs can be used to accelerate tasks such as convolution operations and matrix multiplications, leading to faster inference times compared to CPU-based implementations. In network analysis, FPGAs can process packets at line rate, which is crucial for real-time analytics.

2.5. MTP-1 Overview

The first phase of this M.Tech project (MTP-1) focuses on understanding and setting up the entire system. This includes studying the data flow and the framework of the Processing System (PS) and Programmable Logic (PL) on the PYNQ-Z2 board. The goal is to gain an in-depth understanding of how the PS and PL communicate and interface, which is essential for efficient hardware-software co-design.

During MTP-1, I will implement and test a series of simple applications to develop familiarity with the platform:

- **Adder Module:** A basic arithmetic operation implemented in hardware and interfaced using AXI GPIO.
- **FIR Filter:** A digital signal processing application that demonstrates how FPGAs can be used to accelerate filtering tasks.
- **Image Dimming:** An image processing task that showcases FPGA-based manipulation of image data.
- **MNIST MLP Simulation:** Running a simulation of a multi-layer perceptron (MLP) trained on the MNIST dataset to understand how neural network inference can be mapped to FPGA hardware.

These implementations will help build a strong foundation for more complex tasks and provide insights into the potential and limitations of the PYNQ-Z2 platform.

2.6. MTP-2 Overview

In the second phase of this M.Tech project (MTP-2), the focus will shift to exploring more resource-intensive applications of FPGAs. The primary objective will be to develop an embedded Linux-based platform that functions as an AI-accelerated Network Interface Card (NIC). This platform aims to perform real-time network traffic analytics, enabling efficient monitoring and management.

The FPGA-accelerated NIC will be designed to:

- Enhance responsiveness by offloading intensive computation tasks from the CPU to the FPGA, thus achieving lower latency and higher throughput.
- Provide a scalable solution adaptable to evolving network demands, supporting applications such as anomaly detection, traffic type classification, and Quality of Experience (QoE) prediction.

- Demonstrate the potential of FPGA-based acceleration in machine learning-driven network analytics, showcasing how custom hardware can optimize real-time processing.

The completion of MTP-2 will highlight the advantages of using FPGAs for advanced networking and machine learning applications, solidifying the feasibility of FPGA-based solutions in modern data processing architectures.

Chapter 3

Setup Instructions

3.1. Overview of the PYNQ-Z2 Board and Zynq SoCs

The PYNQ-Z2 board is a versatile development platform designed for Xilinx's Zynq-7000 series SoC (System on Chip). It combines a dual-core **ARM Cortex-A9** processing system (**PS**) with programmable FPGA logic (**PL**). What makes the PYNQ-Z2 unique is its ability to enable developers to program the FPGA using Python, making it an accessible platform for software engineers and hardware developers alike.

Why Use PYNQ-Z2? The PYNQ-Z2 board is widely used in academia and industry for tasks such as machine learning, real-time image and video processing, digital signal processing (DSP), and hardware acceleration of computational tasks. Its Python-based programming environment, using Jupyter Notebooks, allows for a simplified development process when working with the FPGA, without needing to write low-level HDL code. This makes it accessible to a broad range of developers, including those who are not traditional FPGA or hardware engineers.

What is Zynq and Why is it Important? Zynq devices are Xilinx's All Programmable SoCs, which combine the benefits of traditional processors with the flexibility of FPGAs. Unlike regular CPUs that process data sequentially, Zynq SoCs allow parallel processing in the FPGA fabric. This architecture brings hardware acceleration into applications where high-speed performance is critical.

Key differences between Zynq SoCs and traditional processors include:

- **Software-Hardware Co-design:** Zynq devices combine a powerful ARM-based processing system with programmable logic, enabling developers to partition tasks between software and hardware. Traditional processors do not provide this level of hardware customization.
- **Parallelism:** While traditional processors are limited by sequential processing and clock speeds, Zynq SoCs allow multiple tasks to run in parallel in the programmable logic, drastically improving performance for tasks like image processing, machine learning, and DSP.

- **Customization:** FPGA fabric in Zynq SoCs can be customized to match specific application requirements, while traditional processors offer fixed architectures.

Importance of Zynq SoCs Zynq SoCs are essential for applications that require a combination of high-level processing (software running on the ARM cores) and custom hardware acceleration (running on the FPGA fabric). They are used extensively in industries like **aerospace**, **automotive**, **medical devices**, and **communications**, where real-time performance and flexibility are critical.

The PYNQ-Z2 board, built on Zynq-7000, offers a platform to explore these capabilities, providing users with the tools to develop, test, and deploy applications with both high-level software and low-level hardware.

3.2. Setting Up the PYNQ-Z2 Board and Installing Vivado

This section provides a detailed guide on setting up the PYNQ-Z2 development board and installing the Xilinx Vivado Design Suite. It covers hardware assembly, software installation, and configuration steps necessary for developing applications using the PYNQ framework.

3.2.1. Unboxing the PYNQ-Z2 Board and Peripherals

The PYNQ-Z2 board comes with a set of essential components needed to start working with the platform. These include the **power adapter**, the **microSD card**, and the **PYNQ-Z2 board** itself.

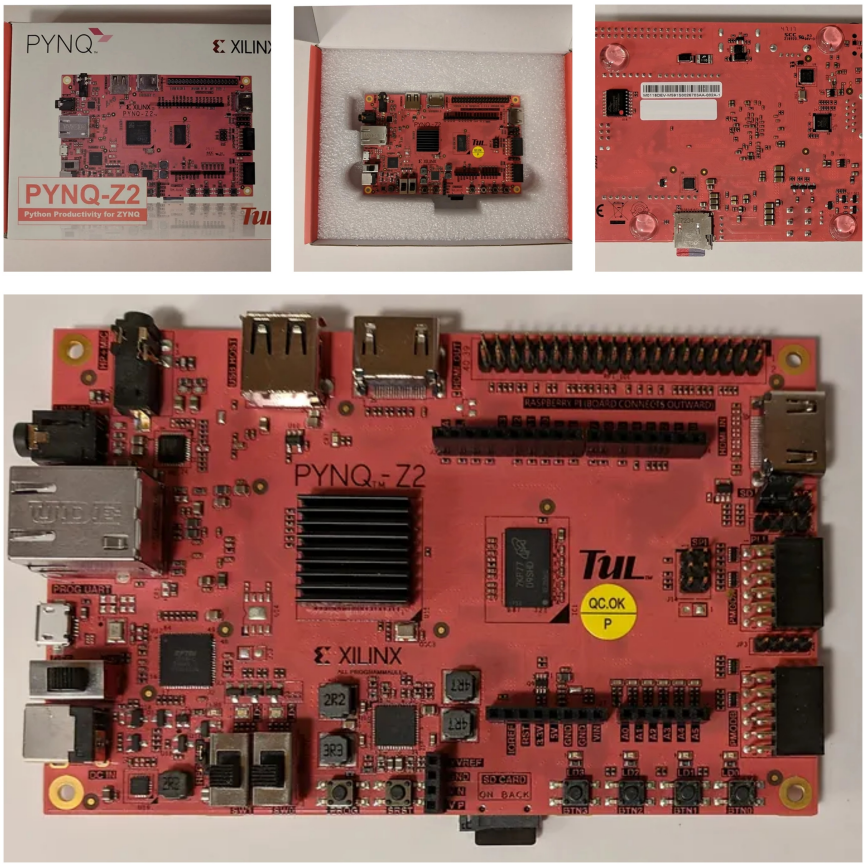

Figure 3.1: Unboxing the PYNQ-Z2 Board and Components

In the collage:

- The top-left image shows the PYNQ-Z2 box, containing the board and other components.
- The top-middle image shows the PYNQ-Z2 board in its protective packaging.
- The bottom images show the front and back sides of the PYNQ-Z2 board.

Peripherals Required: To work with the PYNQ-Z2 board, a few peripherals are required. These include a power adapter, a microSD card, and an Ethernet cable for networking.

Power Adapter:

The power adapter supplies the necessary power to run the PYNQ-Z2 board.

Figure 3.2: Power Adapter for PYNQ-Z2 Board

MicroSD Card:

The microSD card holds the ***PYNQ image***, which is used to boot the board. The board typically comes with a pre-flashed microSD card, or you can download the image from the official website and flash it yourself.

Figure 3.3: MicroSD Card for Booting PYNQ-Z2

3.2.2. Zynq Architecture Overview

The Zynq-7000 devices, like the one used in the PYNQ-Z2 board, are part of Xilinx's All Programmable System on Chip (SoC) family. The Zynq architecture integrates a dual-core ARM Cortex-A9 processor with programmable logic (FPGA) on a single chip. This allows a combination of software programmability (via the ARM cores) with hardware acceleration using FPGA logic.

- **Processing System (PS):** The ARM Cortex-A9 processor handles general-purpose processing tasks, such as running operating systems, networking, and other software applications.
- **Programmable Logic (PL):** The FPGA section allows custom hardware development, which can accelerate specific computational tasks such as signal processing, data encryption, and more.

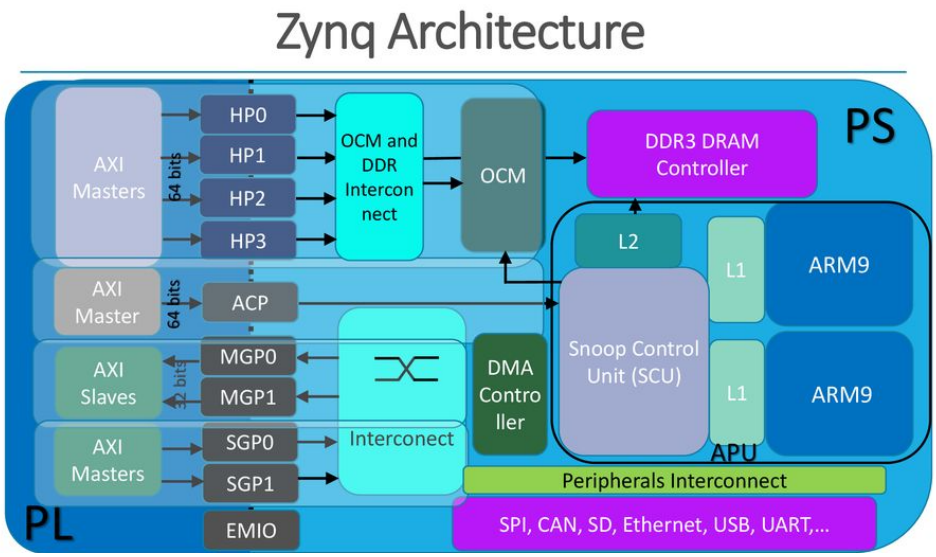

Figure 3.4: Zynq Architecture Overview

Detailed Explanation:

The Zynq architecture is split into two major components: the ***Processing System (PS) and the Programmable Logic (PL)***. The PS includes a dual-core ARM Cortex-A9 processor, which is responsible for running the software, such as the operating system and high-level applications. It has direct access to peripherals, including Ethernet, USB, and DDR memory. The PL is an FPGA fabric that can be customized to accelerate specific tasks, such as video processing, machine learning, or signal processing.

The communication between the PS and PL is handled through the ***AXI Interconnect***, which allows high-speed data exchange. The Zynq-7000 provides multiple AXI interfaces such as High-Performance AXI (**HP**), AXI General-Purpose (**GP**), and AXI Accelerator Coherency Port (**ACP**), which facilitate fast data communication between the processing system and the FPGA fabric.

3.2.3. PYNQ-Z2 Architecture, Internals, and Operating System

The PYNQ-Z2 board is a development platform based on the Zynq-7000 series SoC. It is designed to enable users to program the FPGA using Python and leverage the flexibility of the PYNQ framework. Below are some key features of the board's architecture:

- **Zynq-7000 SoC:** Combines a dual-core ARM Cortex-A9 processor with an FPGA.
- **512MB DDR3 Memory:** Shared between the processing system and programmable logic.
- **HDMI Input/Output:** Allows for real-time video processing.

- **Arduino and Pmod Interfaces:** Allows for expansion with sensors, displays, and other peripherals.
- **Ethernet and USB Ports:** Provides network connectivity and peripheral support.

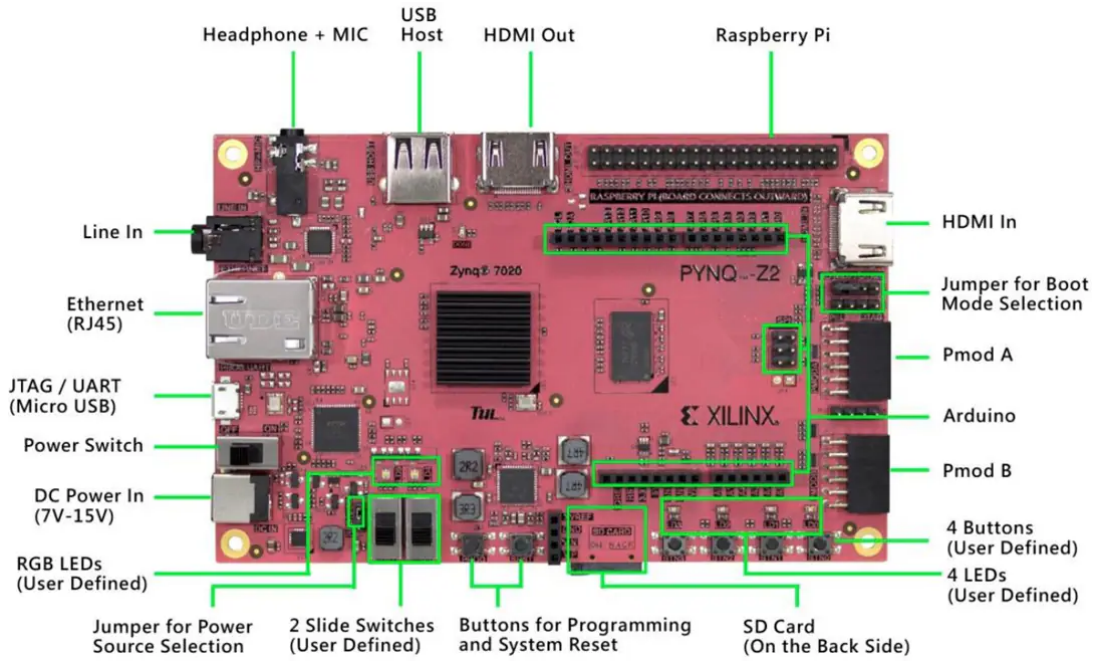

Figure 3.5: PYNQ-Z2 Board Layout

Detailed Explanation of PYNQ-Z2 Internals:

The PYNQ-Z2 board has various connectors and interfaces to interact with external devices and peripherals. Here are some of the key components:

- **GPIO Expansion:** PYNQ-Z2 provides multiple General-Purpose Input/Output (GPIO#) interfaces, compatible with Arduino, Raspberry Pi, and Pmod connectors. These allow easy connection of external components such as LEDs, motors, and sensors.
- **Audio Processing:** The ADAU1761 Audio Codec supports real-time audio capture and processing, making the board suitable for audio effects, sound filtering, and music processing applications.
- **HDMI Input/Output:** The HDMI connectors enable the board to process video data in real-time. This is ideal for applications such as video streaming, image recognition, and real-time signal processing.
- **Ethernet and USB Connectivity:** The board is equipped with a Gigabit Ethernet port for high-speed networking, enabling remote access, data transfer, and cloud connectivity. The USB ports provide support for peripherals such as flash drives, cameras, or WiFi dongles.

PYNQ Operating System (OS):

The PYNQ-Z2 board runs the PYNQ OS, which is built on a lightweight ***Linux-based distribution***. The operating system is specifically designed to support Python-based interaction with the FPGA fabric, offering a simple and powerful interface to control the programmable logic without requiring hardware description languages like VHDL or Verilog.

Here are some key aspects of the PYNQ OS:

- **Kernel and Distribution:** The PYNQ OS is based on ***Ubuntu*** or a similar lightweight Linux distribution, providing essential tools and drivers for the ARM processor and interfacing with the FPGA.
- **Jupyter Notebooks:** The OS includes ***Jupyter Notebooks*** as the main interface for interacting with the board. This enables easy development and debugging using Python code.
- **Pre-installed Libraries:** The OS includes Python libraries for FPGA interaction, including access to hardware overlays, memory-mapped registers, and GPIO control.
- **Remote Access:** The OS allows remote access via **SSH** and the Jupyter Notebook web interface, making it possible to control the board from anywhere on the network.

3.2.4. Setting Up the PYNQ-Z2 Board in Detail

To begin using the PYNQ-Z2 board, a few steps are required, including powering on the board, setting up the SD card with the PYNQ image, connecting the board to a network, and accessing the Jupyter notebook interface.

Powering On and Booting

To power on the PYNQ-Z2 board:

- Insert the microSD card with the PYNQ image into the SD card slot.
- Connect the power adapter to the board.
- Flip the power switch located near the DC power connector.

Once the board is powered on, the system will boot from the microSD card, and you should see the system LEDs light up, indicating successful power-up.

Booting Process Visualization: During booting, the PYNQ-Z2 board first flashes blue lights, as shown in the image below.

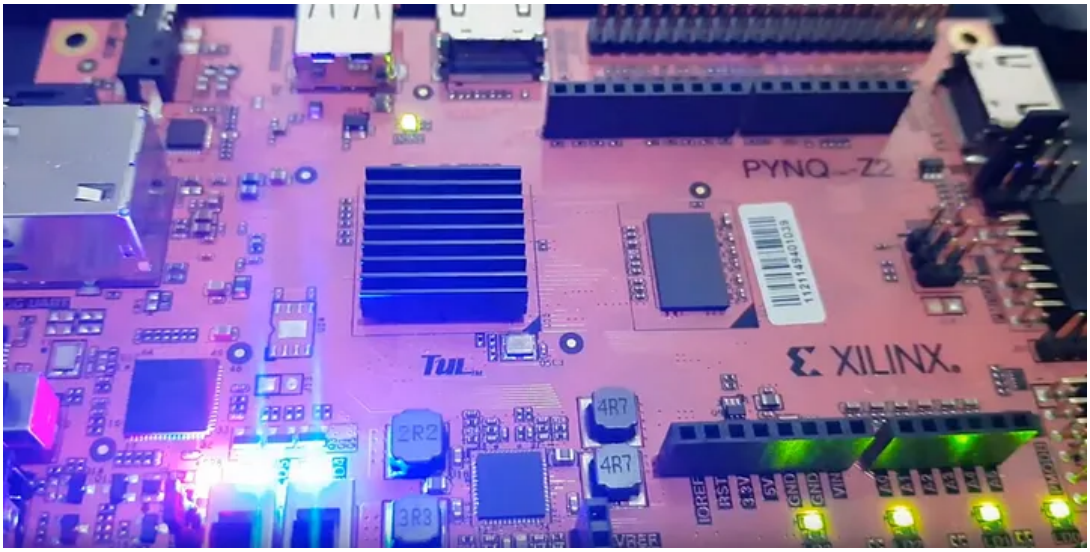

Figure 3.6: PYNQ-Z2 Board Booting up with Blue Lights Flashing

After successful boot-up, the board shows only yellow lights, indicating that it is ready for use.

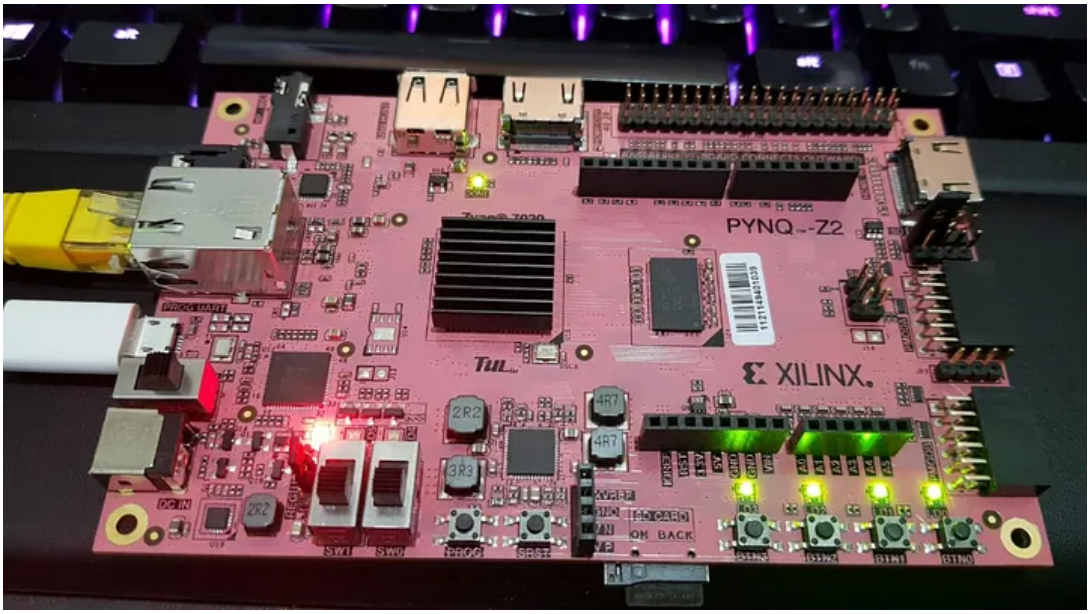

Figure 3.7: PYNQ-Z2 Board Ready with Yellow Lights

Connecting to the Network

Connect the Ethernet port on the PYNQ-Z2 to your local network using an Ethernet cable. This connection is used to access the Jupyter Notebook environment and other services running on the board.

Accessing Jupyter Notebooks

To access the Jupyter Notebooks hosted on the PYNQ-Z2, follow these steps:

- Determine the IP address of the board by logging into your router or using a network scanner such as Angry IP Scanner.
- Once you have the IP address, open a web browser and navigate to: `http://<PYNQ-Z2-IP-Address>/`
- You will see the Jupyter notebook interface where you can run Python programs and interact with the board.

Figure 3.8: Jupyter Notebook Interface on PYNQ-Z2

Terminal Access

The PYNQ-Z2 board also provides a terminal environment where users can execute Linux commands and interact with the board's hardware.

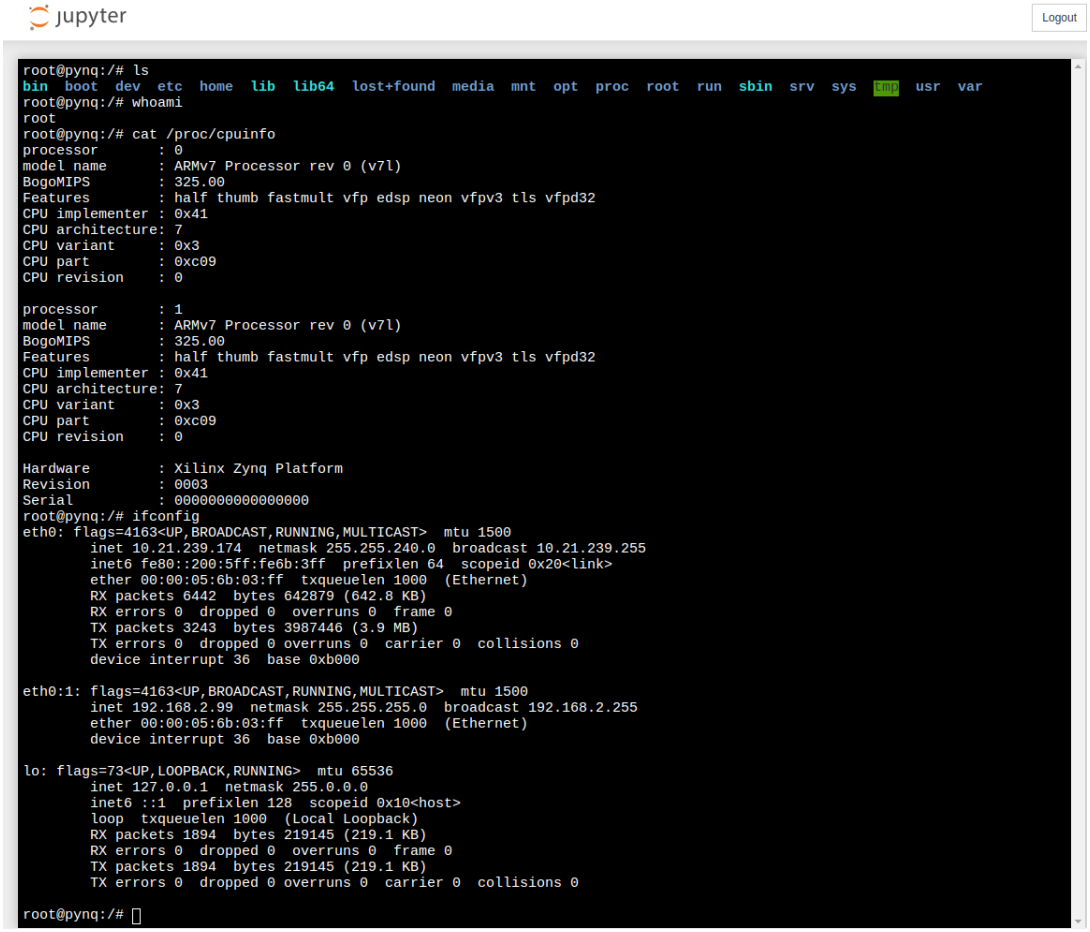

Figure 3.9: Terminal Environment on PYNQ-Z2 Board

3.2.5. Accessing the Board Remotely via SSH

Apart from Jupyter Notebooks, the PYNQ-Z2 board can also be accessed remotely using Secure Shell (SSH). This allows users to interact with the board's terminal directly, enabling them to run Python programs ('.py' files) and manage files on the board without using the web interface.

To connect via SSH, follow these steps:

1. Determine the IP address of your PYNQ-Z2 board.
2. Open a terminal on your computer (Linux or macOS) or a tool like PuTTY (Windows).
3. Use the SSH command to connect to the PYNQ-Z2 board:

```
1       ssh xilinx@<PYNQ-Z2-IP-Address>
```

4. Enter the default password 'xilinx' when prompted.

Once connected, you will have terminal access to the PYNQ-Z2 board. From this terminal, you can manage files, execute commands, and run Python scripts remotely. For example, to run a Python file, you can navigate to the appropriate directory and execute:

```
python3 your_script.py
```

This feature is especially useful when developing and testing scripts outside of the Jupyter environment, or when you prefer working directly through a terminal interface.

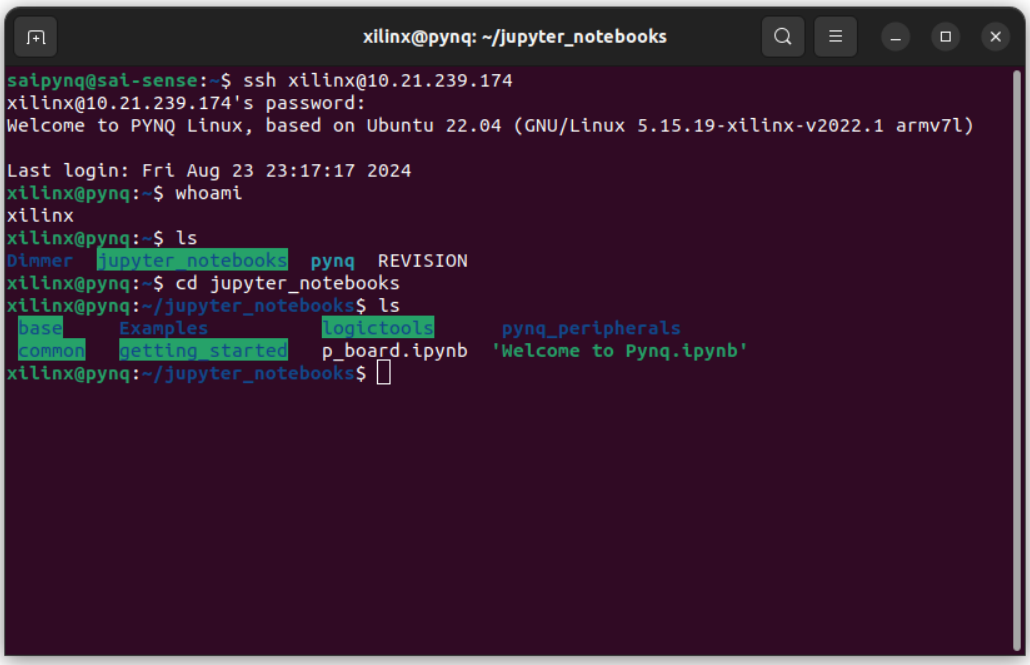

Figure 3.10: SSH Access to PYNQ-Z2 Board Terminal

3.2.6. Testing with a Hello LED Program

To ensure that our PYNQ-Z2 board is properly set up, we can run a simple Python program to control the on-board LEDs. This serves as a basic "Hello World" test to verify the hardware is functioning.

```python
from time import sleep
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")

led0 = base.leds[0] #Corresponds to LED LD0
led1 = base.leds[1] #Corresponds to LED LD1
led2 = base.leds[2] #Corresponds to LED LD2
led3 = base.leds[3] #Corresponds to LED LD3

sw0 = base.switches[0] #Corresponds to SW0
sw1 = base.switches[1] #Corresponds to SW1

while(True):  # All the code below while(True) runs forever
    if (sw0.read() == True):   # Reads SW0 and check if it toggled
        led0.on()              # IF SW0 is ON --> Turn on LED0
        led1.on()              # IF SW0 is ON --> Turn on LED1
    else:
        led0.off()             # ELSE Turn off LED0
        led1.off()             # ELSE Turn off LED1

    if (sw1.read() == True):   # Reads SW1 and check if it toggled
        led2.on()              # IF SW1 is ON --> Turn on LED2
        led3.on()              # IF SW1 is ON --> Turn on LED3
    else:
        led2.off()             # ELSE Turn off LED2
        led3.off()             # ELSE Turn off LED3
```

Figure 3.11: LED Blinking Python Program on PYNQ-Z2

This code will blink the LEDs on the PYNQ-Z2 board based on the status of the onboard switches, providing a quick and easy way to confirm that the board and its GPIO functionality are working correctly.

Output: When both switches are off, the LEDs will be off as shown in the image below:

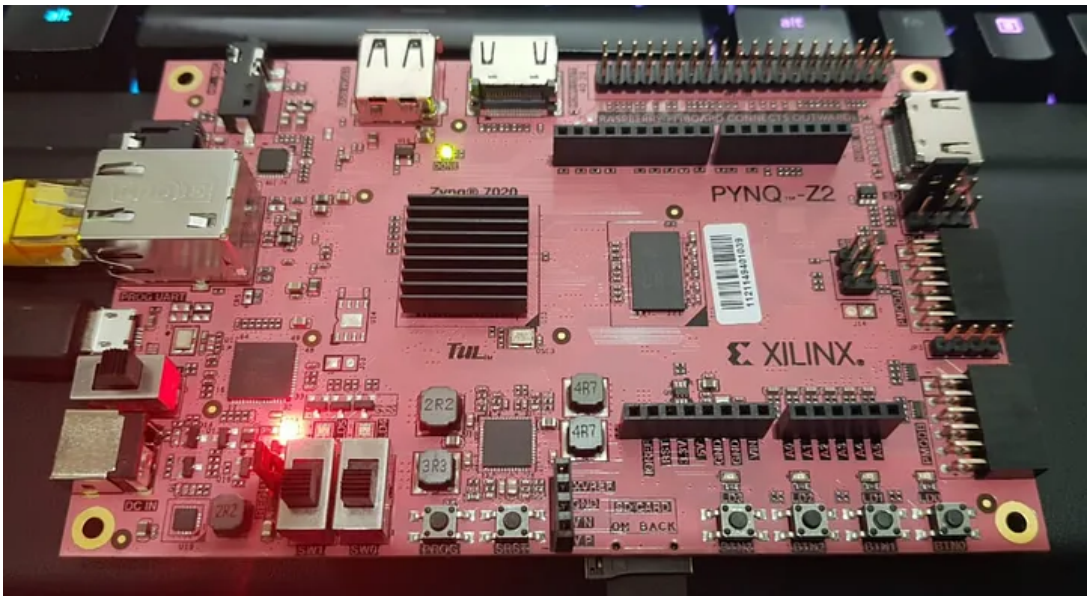

Figure 3.12: Both Switches Off, LEDs Off

When one switch is turned on, the corresponding LEDs will light up as shown below:

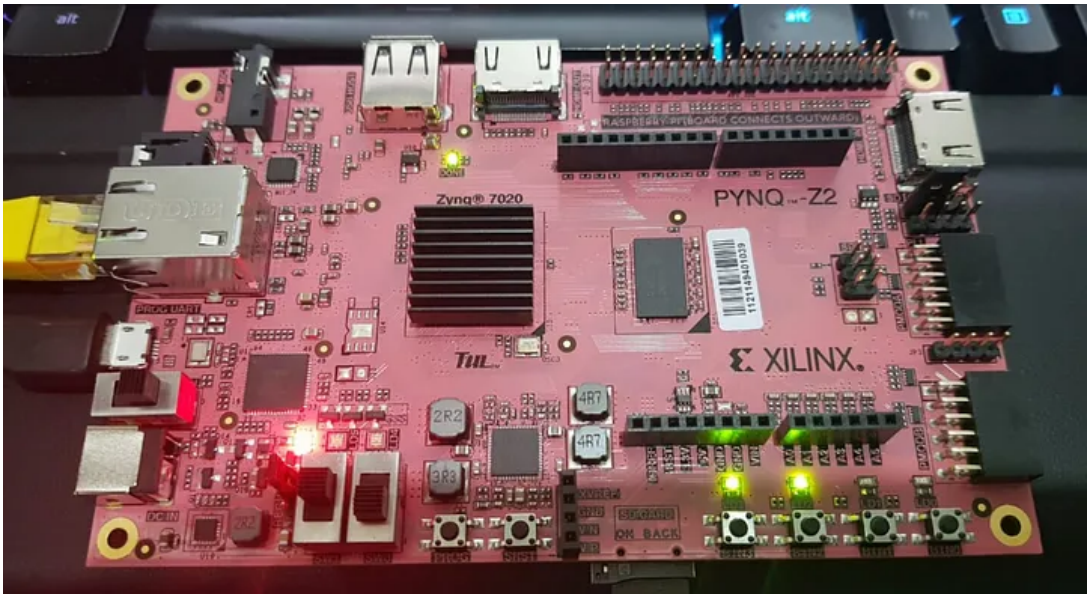

Figure 3.13: Switch 1 On, Corresponding LEDs On

Chapter 4

AXI Interfaces

4.1. Introduction to AXI Interfaces

AXI (***Advanced eXtensible Interface***) is part of the ARM AMBA (***Advanced Microcontroller Bus Architecture***) specification. It is widely used in system-on-chip (SoC) designs to connect various components, such as processors, memory controllers, and peripherals. AXI interfaces support high-performance communication with low-latency and high-throughput characteristics.

4.1.1. Types of AXI Interfaces

The AXI protocol comes in several variations to support different use cases. In this section, we will explore two additional forms of AXI: **AXI Lite** and **AXI Stream**.

4.1.2. AXI Lite Interface

AXI Lite is a simplified subset of the AXI protocol, designed for ***low-bandwidth*** control and ***status register accesses***. It provides a lightweight interface ideal for memory-mapped peripherals and simple register operations. Unlike the full AXI interface, AXI Lite does not support burst transfers or advanced features like out-of-order transactions.

Key characteristics of AXI Lite:

- **Single Transfer Operation**: No burst support, making it suitable for single register accesses.
- **Simpler Protocol**: Fewer signals are required compared to full AXI, which simplifies implementation in low-performance, low-power devices.
- **Ease of Integration**: Commonly used for connecting peripheral registers or control logic in an embedded system.

Applications of AXI Lite:

- **Control Registers**: Ideal for accessing control registers in peripheral devices.

- **Configuration Settings**: Useful for systems where configuration and status readback are needed.

4.1.3. AXI Stream Interface

AXI Stream is another variant of the AXI protocol, optimized for ***high-throughput***, ***point-to-point communication***. It is typically used in data streaming applications such as video, audio, and networking, where data is transferred without addressing overhead.

Key characteristics of AXI Stream:

- **Unidirectional Data Transfer**: Unlike AXI Lite, AXI Stream is designed for streaming data from one source to a sink (point-to-point).
- **No Addressing**: Since there is no need for address information in streaming applications, AXI Stream omits address/control channels.
- **High Throughput**: Supports continuous data flow, making it ideal for applications like video processing, image capture, and high-speed networking.

Applications of AXI Stream:

- **Video Processing Pipelines**: Used in video processing systems to stream frames between different processing stages.
- **Audio and Networking Systems**: Provides an efficient means of moving large quantities of data continuously without interruption.

4.2. Real-World Examples of AXI Interface Implementations

AXI interfaces are used in a wide range of industries, from embedded systems to high-performance computing. Below are some real-world examples where AXI is used effectively to improve performance and enable complex data transfers.

4.2.1. FPGA-Based Image Processing System

In an FPGA-based image processing system, AXI Stream is used to transfer frames of video between different FPGA-based image processing blocks. Each block performs specific tasks, such as filtering, edge detection, or compression, using AXI Stream to pass the video frames from one block to the next in real-time.

4.2.2. Embedded Control System

In embedded control systems, AXI Lite is used to connect the main processor to a series of control registers within peripherals, such as sensors and actuators. The simplicity of AXI Lite allows for quick, low-latency access to configuration registers, while still supporting the memory-mapped addressing mode.

4.2.3. High-Speed Networking System

In high-speed networking systems, AXI Stream is employed to move packets of data from a network interface card (NIC) to the processor. The continuous data transfer enabled by AXI Stream allows the network to achieve maximum throughput, handling large bursts of data without buffering delays.

4.3. AXI GPIO and AXI DMA Interfaces

AXI also includes specialized interfaces such as AXI GPIO and AXI DMA, which provide additional functionality in system designs.

4.3.1. AXI GPIO Interface

The AXI GPIO (General Purpose Input/Output) interface allows an AXI-based system to interact with general-purpose inputs and outputs. Below is the configuration of the AXI GPIO IP and its respective block diagram.

Figure 4.1: AXI GPIO IP

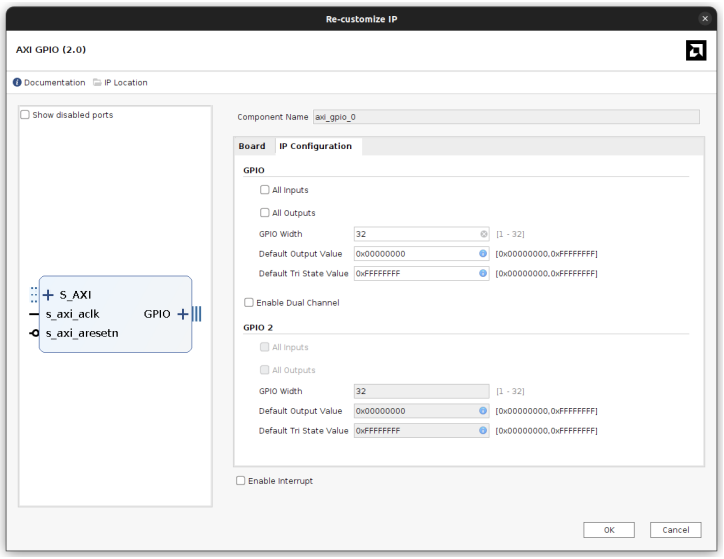

Figure 4.2: AXI GPIO Configuration Interface

Key features of AXI GPIO:

- **All Inputs/All Outputs**: These options configure the direction of the GPIO pins. When set to "All Inputs," the GPIO pins are configured to read signals from external devices, such as switches or sensors. When set to "All Outputs," the pins are configured to output signals, which is useful for controlling external devices like LEDs.
- **Enable Dual Channel**: When enabled, this option allows you to have two independent channels of GPIOs. One channel could be used for inputs, while the other could be used for outputs. This provides flexibility in interfacing with different peripherals.
- **GPIO Width**: This defines how many GPIO pins are available. For example, if the width is set to 32, there will be 32 GPIO pins that can be controlled as inputs or outputs.
- **Default Output Value/Default Tri-State Value**: These parameters set the initial value of the GPIOs at system startup or reset. The default output value specifies the default logic level of the GPIO pins. The default tri-state value sets whether the pin is in a high-impedance state, which is useful when the pin needs to be disconnected from the circuit.
- **Enable Interrupt**: This option allows the GPIO to trigger interrupts, enabling real-time system responses based on external events like button presses.

Applications of AXI GPIO:

- **Interfacing with Sensors and Actuators**: AXI GPIO is used to connect the system to external hardware, such as sensors and actuators, for control and data acquisition.
- **Control of LEDs and Buttons**: AXI GPIO can be used for simple I/O tasks, such as toggling LEDs or reading button states.

4.3.2. AXI DMA Interface

AXI DMA (Direct Memory Access) enables high-performance data transfer between memory and peripherals. Below is the configuration of the AXI DMA IP and its respective block diagram.

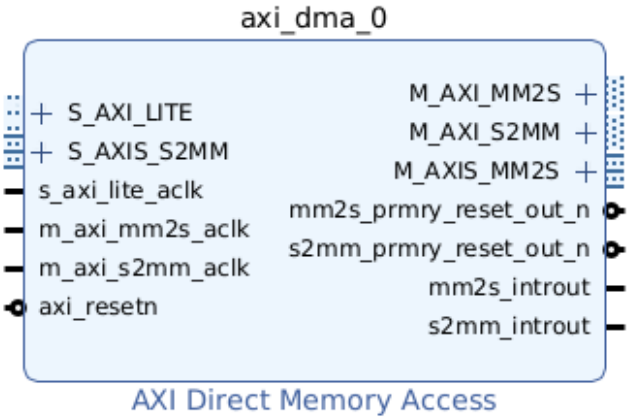

Figure 4.3: AXI DMA IP

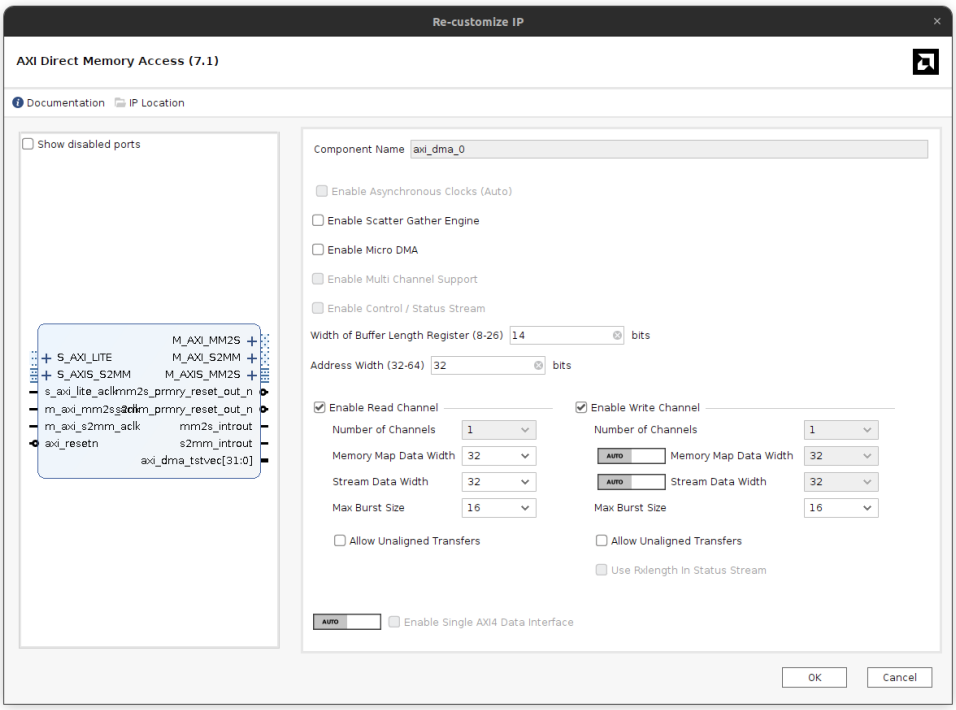

Figure 4.4: AXI DMA Configuration Interface

Key features of AXI DMA:

- **Scatter-Gather Engine**: This feature allows AXI DMA to transfer data from non-contiguous memory locations. The scatter-gather engine is crucial for handling fragmented memory regions, which may occur in complex systems like video buffers or networking applications.
- **Micro DMA**: This enables a lightweight, low-power DMA that is optimized for systems requiring reduced overhead. It provides basic DMA functionality without the power and resource demands of a full DMA engine.

- **Read Channel and Write Channel**: These channels define how data is transferred between memory and peripherals. The read channel handles memory-to-peripheral transfers, and the write channel handles peripheral-to-memory transfers. Both channels support high-throughput data transfer for applications like video streaming or network packet handling.
- **Data Width**: The data width parameters specify the width of the data bus. For instance, a 32-bit data width allows the DMA to transfer 32 bits of data in one cycle. A wider data bus can increase throughput, but at the cost of additional resource usage.
- **Max Burst Size**: The burst size refers to the number of data transfers that occur in one transaction. A larger burst size reduces the overhead caused by frequent bus arbitration, improving throughput for applications requiring continuous data flow, such as audio or video.
- **Enable Interrupt**: This option triggers an interrupt when the data transfer is complete, allowing the processor to handle the next task or process the data.

Applications of AXI DMA:

- **Video and Audio Data Transfers**: AXI DMA is used in applications where high-speed data transfers are required, such as video and audio streaming.
- **Networking Systems**: AXI DMA can be employed to transfer large network data packets between memory and network interfaces without CPU intervention.

4.4. AXI Transaction Details: Ordering, Burst Types, and Error Responses

In this section, we will explore some advanced features of AXI, including transaction ordering, burst types, and error responses.

4.4.1. Transaction Ordering

AXI allows out-of-order transaction completion. This means that transactions initiated later can complete before earlier transactions, as long as no dependency exists between them. AXI achieves this by separating the address and data phases, enabling efficient overlapping of operations.

Out-of-order transactions improve overall throughput in systems where latency is critical and can be hidden behind other operations.

4.4.2. Burst Types

AXI supports three types of burst transfers:

- **Fixed Burst**: The address remains constant throughout the burst. This is typically used for streaming data to a fixed location.

- **Incremental Burst**: The address is incremented after each data transfer. This is the most common type, used when transferring data between memory and peripherals.
- **Wrap Burst**: After reaching a defined boundary, the address wraps around to an earlier value. This is useful for transferring circular buffers.

4.4.3. Error Responses

AXI provides mechanisms to handle errors during transactions. Two types of responses can be returned by the slave in case of failure:

- **SLVERR (Slave Error)**: Indicates that the slave encountered an issue with the transaction, such as attempting to access a protected memory region.
- **DECERR (Decode Error)**: Indicates that the address provided by the master did not match any valid slave or memory address range, typically signaling an invalid address.

Chapter 5

Adder with AXI GPIO

5.1. Designing and Implementing an Adder on PYNQ-Z2 using Vivado

This tutorial provides a detailed step-by-step guide for designing and implementing an adder on the PYNQ-Z2 board using Vivado. We will first demonstrate how to integrate the default Adder IP with an AXI GPIO interface, followed by the creation and integration of a custom Verilog-based Adder IP.

5.1.1. Adder with Default IP

In this project, we will create an AXI GPIO interface, one of the simplest AXI interfaces, to handle communication between the Zynq Processing System and the adder using the default Adder IP. The design will be controlled via a Jupyter notebook, allowing us to interact with the adder and verify its functionality through the PYNQ-Z2 platform.

Vivado Block Design

The first step is to create the block design in Vivado. We use the default Adder/-Subtractor IP and connect it to the Zynq Processing System via AXI GPIO blocks. The following diagram illustrates the block design:

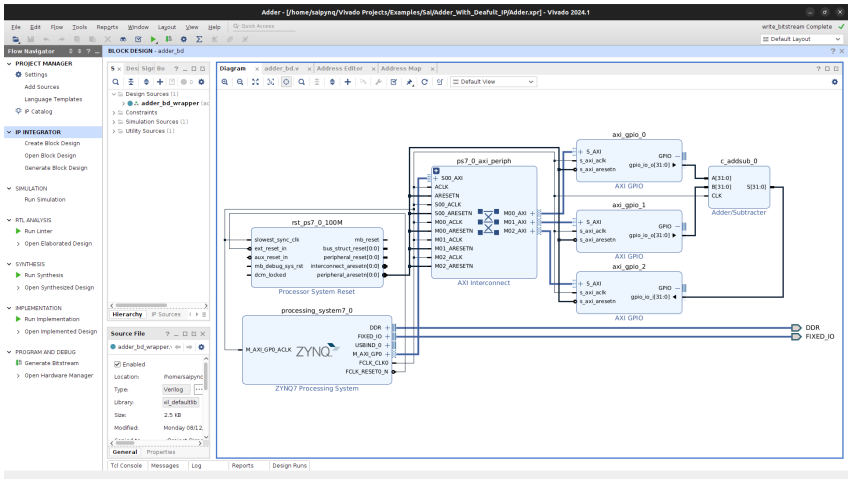

Figure 5.1: Vivado Block Design for Simple Adder

Steps for Creating the Block Design:

1. **Create a New Block Design:**
   - In Vivado's `IP Integrator`, click on `Create Block Design`.
   - Provide a name for the block design (e.g., `adder_bd`).
   - The block design canvas will open, allowing you to start adding components.
2. **Add Zynq Processing System:**
   - From the `IP Catalog`, drag and drop the `Zynq Processing System`.
   - Run `Block Automation` to automatically configure the processing system with AXI interfaces.
   - Make sure at least one AXI General-Purpose (GP) Master Interface is enabled.
3. **Add Adder/Subtractor IP:**
   - From the `IP Catalog`, search for `Adder/Subtractor` and add it to the block design.
   - This IP performs the addition operation using operand inputs provided by the AXI GPIO blocks.
4. **Add AXI GPIO Blocks:**
   - Add three instances of the `AXI GPIO` IP block from the `IP Catalog`.
   - These GPIO blocks will interface with the Zynq Processing System and the Adder IP to handle input and output data.
5. **Connect the IP Blocks:**
   - Connect the GPIO outputs (operands) to the Adder/Subtractor IP's input ports (A and B).
   - Connect the Adder/Subtractor output port to the third GPIO block (for reading the result).
6. **Configure the AXI GPIO Blocks:**

– Set the GPIO width to 32 bits (since the adder operates on 32-bit data).
– Configure the first two GPIOs as `Outputs` (for operand A and B) and the third as `Input` (for the result).

AXI GPIO Configuration

The AXI GPIO block in Vivado is used to interface with general-purpose input/output signals, and in this design, the GPIO is used to send operands to the Adder IP and receive the result. The following image shows the configuration window for the AXI GPIO:

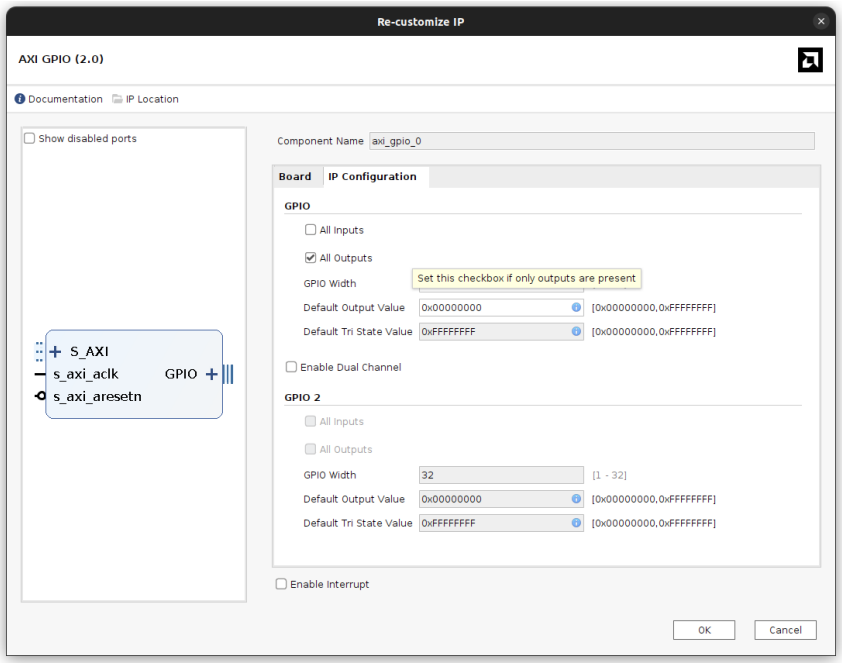

Figure 5.2: AXI GPIO Configuration for the Adder

Steps for Configuring AXI GPIO:

(a) **Select GPIO Direction:**
  – For the first two GPIO blocks, set the direction to `All Outputs`. This configures them to send operands (A and B) from the processor to the adder.
  – Set the third GPIO block to `Input`, which will receive the result from the Adder IP.

(b) **Set GPIO Width:**
  – Set the GPIO width to 32 bits, which matches the Adder IP's input width.

(c) **Configure Default Values:**
  – You can configure the default values for the GPIO outputs and tri-state values if needed, but they can remain at their defaults for this design.

Once the configuration is complete, save the design and proceed with generating the bitstream.

7. **Run Connection Automation:**
   - Use `Connection Automation` to automatically wire the AXI interfaces.
   - Once done, validate the design.

Once the configuration is complete, save the design and proceed with generating the bitstream.

Generating the Bitstream

Generating the bitstream involves several steps in Vivado, which includes creating the HDL wrapper, running synthesis, and generating the bitstream itself. Follow the steps below:

Steps to Generate the Bitstream:

1. **Create HDL Wrapper:**
   - In the `Sources` tab, right-click on the block design (e.g., `adder_bd.bd`) and select `Create HDL Wrapper`.
   - Choose `Let Vivado manage wrapper and auto-update`.
   - This step generates the necessary Verilog or VHDL file that describes the top-level module of the design, which will be used for synthesis.
2. **Run Synthesis:**
   - Once the HDL wrapper is created, right-click on the project name in the `Flow Navigator` pane and select `Run Synthesis`.
   - Synthesis is the process where Vivado converts the RTL (Verilog or VHDL) into a gate-level netlist.
   - After synthesis completes, a window will appear asking if you want to `Open Synthesized Design`. You can choose to open it to inspect the design, but this is optional.
3. **Run Implementation:**
   - After synthesis, run implementation by right-clicking on the project name and selecting `Run Implementation`.
   - Implementation maps the netlist onto the physical resources of the FPGA (LUTs, Flip-Flops, etc.).
   - When the implementation is complete, you can choose to open the implemented design to inspect the placement and routing of the resources.
4. **Generate the Bitstream:**
   - After implementation is complete, right-click the project name and select `Generate Bitstream`.
   - Vivado will take the implemented design and generate a `.bit` file that can be used to program the FPGA.
   - Once the bitstream generation is complete, you will be prompted to `Open Hardware Manager`.
5. **Program the FPGA:**

- Connect your PYNQ-Z2 board to the host computer using a JTAG cable or USB.
- Open the Hardware Manager, connect to the target, and program the FPGA using the generated bitstream.

After the bitstream has been generated, it can be loaded onto the FPGA via the Jupyter notebook as described in the next section.

5.1.2. Jupyter Notebook Directory Structure

When working with the PYNQ framework, it is essential that the names of the `.bit` file and the corresponding `.hwh` file are the same. The `.bit` file contains the bitstream, which configures the FPGA hardware, while the `.hwh` file contains hardware metadata, describing the hardware's structure and AXI interfaces.

By having matching names (e.g., `adder.bit` and `adder.hwh`), the PYNQ framework can automatically load the correct hardware description when the bitstream is loaded. If the names do not match, PYNQ will not be able to associate the two files, resulting in potential errors when accessing the hardware interfaces.

The following image shows a directory listing in Jupyter containing the `.bit`, `.hwh`, and `.ipynb` files used for this design:

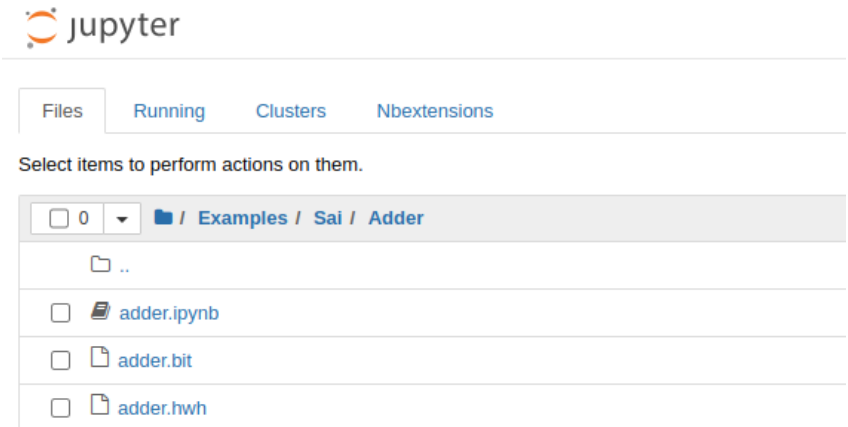

Figure 5.3: Directory Listing in Jupyter Showing `.bit`, `.hwh`, and `.ipynb` Files

5.1.3. Jupyter Notebook Implementation

After generating and programming the bitstream onto the FPGA, we use a Jupyter notebook (via the PYNQ framework) to interact with the adder design. This allows us to write the operands via GPIO, trigger the addition, and read the result.

Figure 5.4: Jupyter Notebook Code for Controlling the Adder

Steps in the Jupyter Notebook:

1. **Load the Bitstream:**

   ```
   from pynq import Overlay
   overlay = Overlay('adder.bit')
   ```

   This command loads the bitstream into the FPGA, configuring the adder design.

2. **Configure GPIOs:**

   ```
   from pynq import MMIO
   gpio_0_baseaddr = 0x41200000
   gpio_1_baseaddr = 0x41210000
   gpio_0 = MMIO(gpio_0_baseaddr, 0x10000)
   gpio_1 = MMIO(gpio_1_baseaddr, 0x10000)
   ```

   These commands configure the memory-mapped I/O (MMIO) for the GPIO blocks. This allows the notebook to send and receive data through the GPIOs.

3. **Write Data to GPIO:**

   ```
   gpio_0.write(0x00, 0x00000053)  # Operand A (0x53)
   gpio_0.write(0x08, 0x00000036)  # Operand B (0x36)
   ```

   These values (0x53 and 0x36) will be sent as operands to the adder.

4. **Read the Result:**

   ```
   result = gpio_1.read(0x00)
   print(f"Result: {hex(result)}")
   ```

   The result of the addition will be 0x89 (hex), which is the sum of 0x53 and 0x36.

Explanation of GPIO Base Addresses:

The base addresses used in the Jupyter notebook for the AXI GPIOs come from the **Address Editor** in Vivado. Each AXI GPIO block is assigned a unique memory-mapped base address. The Zynq Processing System (PS) communicates with the GPIO peripherals by reading from or writing to these specific memory locations.

In the design shown in the Address Editor, the assigned base addresses are as follows:

- **AXI GPIO 0 Base Address:** `0x41200000`
- **AXI GPIO 1 Base Address:** `0x41210000`
- **AXI GPIO 2 Base Address:** `0x41220000`

These addresses are mapped in the Address Editor, as shown in the following screenshot:

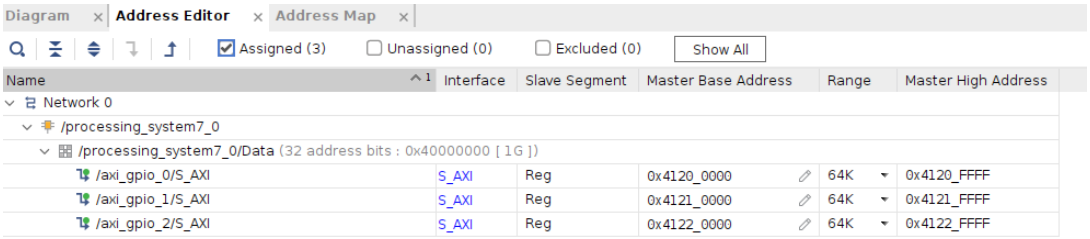

Figure 5.5: Address Map from Vivado for AXI GPIO Blocks

The base addresses are used to access the GPIO registers in the Jupyter notebook via the `MMIO` class, which enables Python to communicate with the hardware. Each address corresponds to the control and data registers of the AXI GPIO peripherals, allowing the software to interact with the hardware design efficiently.

5.1.4. Implementing a Custom Adder IP on PYNQ-Z2 using Vivado

In this section, we will implement a custom adder IP in Vivado. Unlike using the default Adder IP, we will create our own Verilog-based Adder module, package it as a custom IP, and integrate it into the Vivado block design. This allows for a more flexible and educational approach to hardware design.

Custom Verilog Code for the Adder IP

The following Verilog code defines a simple 32-bit adder. It takes two 32-bit inputs, adds them, and produces a 32-bit output. The screenshot below shows the code:

```
`timescale 1ns / 1ps

module adder (
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);
    assign sum = a + b;
endmodule
```

Figure 5.6: Verilog Code for the Custom 32-bit Adder

The module is defined as follows:

- **Inputs:** Two 32-bit inputs, 'a' and 'b', are passed to the module.
- **Output:** A 32-bit output, 'sum', is produced by adding the two inputs.
- **Simple Operation:** The Verilog 'assign' statement is used to continuously compute the sum of 'a' and 'b', with the result assigned to 'sum'.

This custom Adder IP is wrapped in a Verilog module and will be packaged as an IP in Vivado.

Adding the Custom Adder IP to the Block Design

Once the custom adder Verilog code has been written, the next step is to package it as an IP and add it to the Vivado block design. The following steps describe how to add the custom IP to your block design:

Steps for Adding Custom Adder IP:

1. **Package the Adder as an IP:**
   - In Vivado, go to `Tools > Create and Package IP`.
   - Select `Create a new AXI peripheral` or use the RTL module.
   - Choose the Verilog file containing the custom adder as the source file.
   - Follow the steps to define the IP's interfaces and ports. Ensure that you expose the input and output ports for integration with AXI GPIO.
2. **Add the Custom Adder IP to the Block Design:**
   - After packaging the IP, it will be available in the `IP Catalog`.
   - Add the custom Adder IP to the block design in the same way as any standard IP.
   - Integrate it with AXI GPIO blocks as shown in the diagram below.

Block Design for Custom Adder IP

The block design integrates the custom adder with AXI GPIO for input and output handling. In this design, the Zynq Processing System communicates with the adder through the GPIO interface. The following diagram illustrates the block design:

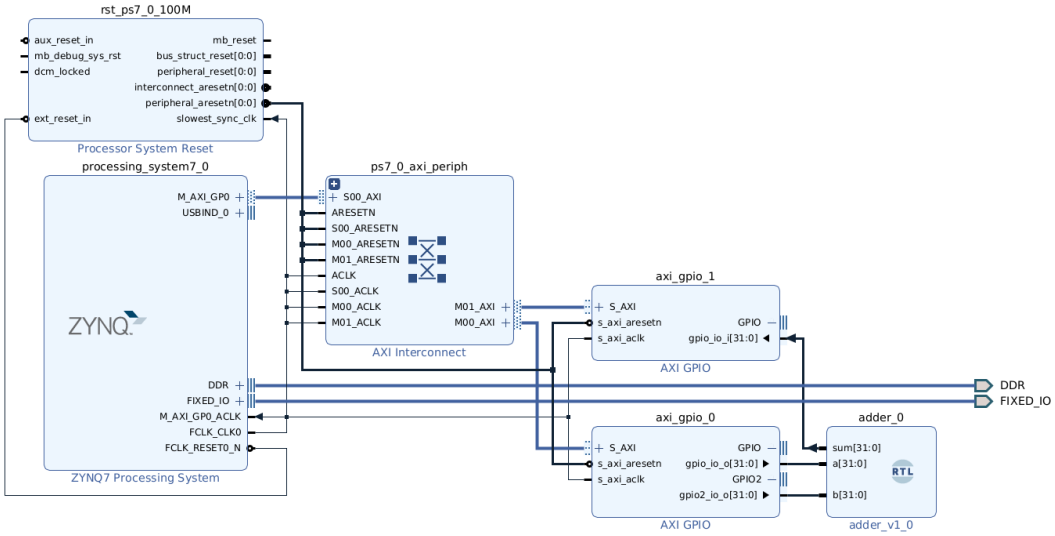

Figure 5.7: Vivado Block Design Integrating the Custom Adder IP

Key Components of the Design:

- **Zynq Processing System:** The central controller of the design, handling AXI communication and data processing.
- **AXI GPIO:** Two GPIO blocks are used to handle input and output. The first GPIO sends the operands (‘a‘ and ‘b‘) to the adder, while the second GPIO reads the result (‘sum‘) from the adder.
- **Custom Adder IP:** This block is the custom adder created using Verilog. It interfaces with the GPIO to perform the addition operation on 32-bit values.

Steps for Connecting Custom Adder in Block Design:

- **Input Connection:** The GPIO block is connected to the custom adder’s inputs (‘a‘ and ‘b‘), allowing the Zynq Processing System to pass operands.
- **Output Connection:** Another GPIO block reads the output (‘sum‘) from the adder and sends it back to the processing system.

By following these steps, we will have successfully integrated a custom adder into the PYNQ-Z2 block design, demonstrating how to extend the functionality of the PYNQ framework with user-defined hardware.

Chapter 6

Adder with AXI DMA

6.1. Implementing Adder using AXI DMA on PYNQ-Z2

In this tutorial, we will implement an adder using the AXI Direct Memory Access (DMA) interface for high-speed data transfer between the Zynq Processing System and the custom Adder IP. This implementation demonstrates the usage of AXI DMA for moving data from system memory to the hardware IP, performing the addition operation, and returning the result.

While DMA may not be strictly necessary for an application like this, where we are only transferring a small amount of data, it is used here to showcase how AXI DMA can be configured for more advanced data transfer scenarios. In practice, **DMA is highly effective for large datasets and high-throughput applications**. For small, simple operations like this adder, using AXI GPIO could suffice, but this exercise will show how to interface DMA for learning purposes.

6.1.1. DMA Configuration Details

Before setting up the block design, configuring the AXI DMA IP correctly is essential. The following image shows the detailed configuration window of the AXI DMA IP:

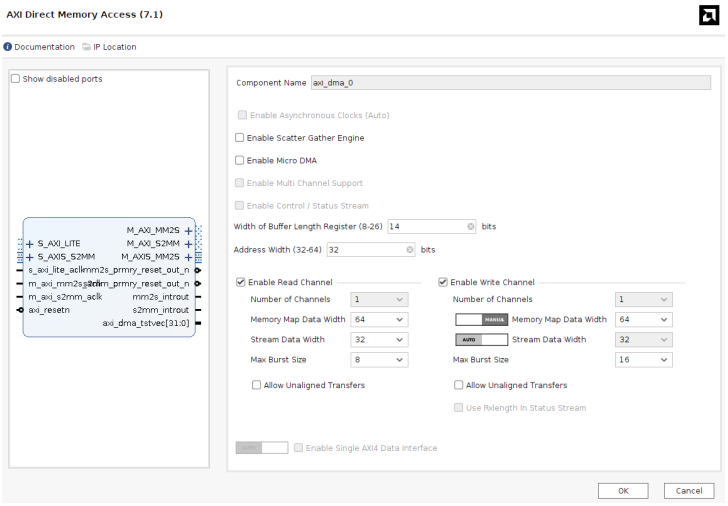

Figure 6.1: AXI DMA Configuration Window

Key Configuration Settings:

- **Disable Scatter Gather Engine and Micro DMA:** These are unnecessary for simple streaming applications like this adder. The Scatter Gather Engine allows for non-contiguous memory transfers, which is useful for more complex data movement, but here, we are working with a simpler linear buffer.
- **Read and Write Channels:** Both channels should be enabled, as we need to send data (write) to the custom adder IP and receive data (read) back from it. The `M AXI MM2S` (***Memory Map to Stream***) channel transfers data from memory to the IP, while the `S AXI S2MM` (***Stream to Memory Map***) channel transfers data from the IP back to memory.
- **Memory Map Data Width:** Set this to 64 bits to match the memory access width of the Zynq processing system. It's important to choose a memory width that matches the system's architecture to avoid misalignment and potential performance issues.
- **Stream Data Width:** Set this to 32 bits, matching the width of the data processed by the custom adder. The stream width must align with the input/output width of your IP to avoid data corruption or misalignment.
- **Max Burst Size:** The burst size determines how many bytes can be transferred in a single transaction. For this example, we can set it to 8 or 16, but in more data-intensive applications, larger burst sizes can significantly improve throughput.

Important Considerations: When using AXI DMA, always ensure that the data widths between our custom IP and the DMA are correctly matched. Any mismatch could result in incorrect data transfers. Also, disabling unnecessary features like Scatter Gather and Micro DMA reduces complexity for simple applications.

6.1.2. Block Design for AXI DMA Interface

After configuring the DMA, the next step is to create the block design in Vivado using the AXI DMA IP. The DMA enables efficient data transfer between memory

and the custom adder without relying on CPU involvement, allowing us to offload the data movement task. Below is the block design we will use:

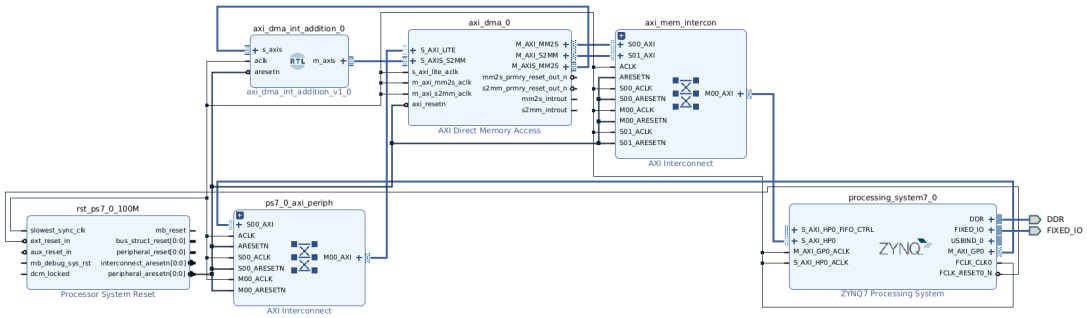

Figure 6.2: Block Design for AXI DMA Interface with Custom Adder

Steps for Creating the Block Design:

1. **Add Zynq Processing System:**
   - In Vivado's `IP Integrator`, create a new block design and add the `Zynq Processing System`.
   - Run `Block Automation` to configure the PS with AXI interfaces. Ensure to enable the High-Performance (HP) AXI ports, which provide higher bandwidth and are necessary for DMA transactions.
   - The `M AXI GP0` interface is connected to the custom adder IP to handle the memory-mapped registers of the adder.
   - Enable `HP0` ports on the PS for high-speed data transfer, as these ports are optimized for use with DMA and large data transactions.
2. **Add AXI DMA IP:**
   - From the `IP Catalog`, add the `AXI DMA` IP to the block design.
   - Make sure to connect the `S_AXIS_S2MM` (Stream to Memory Map) and `M_AXIS_MM2S` (Memory Map to Stream) interfaces of the DMA to the custom IP for data streaming.
   - The `S AXI Lite` interface of the DMA is connected to the `M AXI GP0` port of the PS to configure the DMA via software.
3. **Add Custom Adder IP:**
   - Integrate the custom adder IP that performs the addition operation. The input and output ports of the adder should be connected to the AXI DMA.
   - The AXI Lite interface of the adder is connected to the `M AXI GP0` port of the PS for register access. This interface allows you to configure and control the IP.
4. **Configure the Interconnect:**

– Use an AXI Interconnect IP to connect the master (Zynq PS) to the slaves (AXI DMA and custom adder). This manages the multiple master-slave connections and ensures smooth data flow between the blocks.
– The interconnect provides a flexible and scalable way to connect different AXI-based IPs. In this design, it facilitates communication between the processing system, the DMA, and the adder.

Important Design Considerations: When configuring the interconnect, make sure the clock signals and reset signals are properly aligned across the components. The DMA, custom IP, and Zynq PS must operate on synchronized clocks to avoid timing issues. Additionally, ensure proper configuration of burst sizes for DMA to maximize throughput.

6.1.3. AXI Interconnect and Connections

The AXI Interconnect is a crucial component in this design. It routes data between the Zynq PS, AXI DMA, and the custom adder. The interconnect facilitates multiple ***master-slave connections***, such as DMA to system memory and AXI to the custom IP.

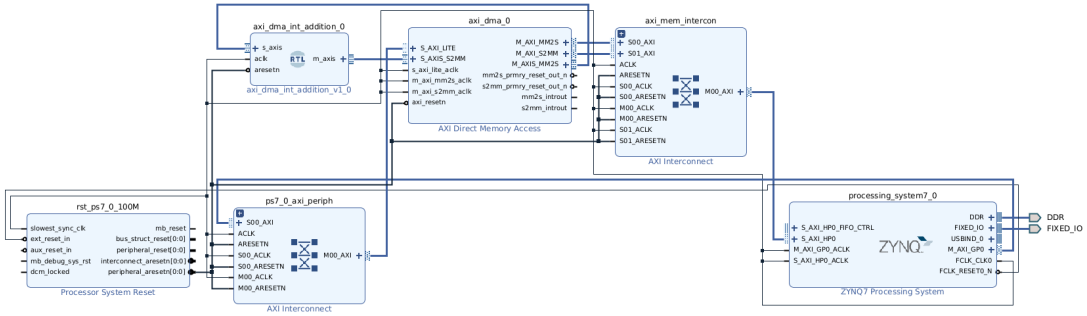

Figure 6.3: Block Design showing key AXI connections between Zynq PS, DMA, and custom Adder IP

Key Connections:

– **Master-Slave Connections:** The Zynq PS acts as the master, with AXI DMA and the custom adder IP as slaves. The AXI Interconnect is placed between the PS and the IPs, managing data flow. ***The `M AXI GP0` interface handles the control register access, while the `HP` ports handle high-speed data transactions.***
– **HP Interfaces:** The Zynq PS's High-Performance (HP) AXI interfaces provide a fast communication link with the system memory, which is essential for DMA data transfers. The HP ports are optimized for high-bandwidth, low-latency communication. Here, the `M AXI MM2S` and `S AXI S2MM` channels are connected to the HP0 port for efficient data streaming.

– **AXI DMA Connections:** The `M AXI MM2S` (Memory Map to Stream) and `S AXI S2MM` (Stream to Memory Map) interfaces handle the streaming data to and from the custom adder IP. These connections are crucial for moving data between the system memory and the hardware IP. The `S AXI Lite` connection allows DMA configuration from the Zynq PS.

Important Considerations: Ensure that the clock domains of the Zynq PS, AXI DMA, and the custom adder are properly synchronized. Mismatched clock domains could lead to data corruption or transfer delays. Additionally, using the HP ports ensures efficient data transfer, which is especially important in data-intensive applications.

6.1.4. Verilog Code for Custom AXI DMA-based Adder IP

Below is the Verilog code for a custom AXI-Stream-based adder that uses AXI DMA for transferring two integers to the IP, performs the addition, and sends the result back.

Listing 6.1: Verilog Code for AXI DMA-based Adder IP

```
module axi_dma_int_addition #
(
    parameter integer DATA_WIDTH = 32
)
(
    // AXI-Stream input (Slave)
    input wire [DATA_WIDTH-1:0] s_axis_tdata,
    input wire s_axis_tvalid,
    output reg s_axis_tready,
    input wire s_axis_tlast,

    // AXI-Stream output (Master)
    output reg [DATA_WIDTH-1:0] m_axis_tdata,
    output reg m_axis_tvalid,
    input wire m_axis_tready,
    output reg m_axis_tlast,

    // Clock and Reset
    input wire aclk,
    input wire aresetn
);

// State machine for controlling data flow
localparam IDLE = 2'b00,
           ADD  = 2'b01,
           SEND = 2'b10;

reg [1:0] state;
reg [DATA_WIDTH-1:0] int_a, int_b, sum;  // Internal registers for
   the integers and their sum

always @(posedge aclk) begin
    if (!aresetn) begin
        m_axis_tvalid <= 1'b0;
        m_axis_tdata <= 0;
```

```
35          m_axis_tlast <= 1'b0;
36          state <= IDLE;
37          int_a <= 0;
38          int_b <= 0;
39          sum <= 0;
40      end else begin
41          case (state)
42              IDLE: begin
43                  if (s_axis_tvalid) begin
44                      int_a <= s_axis_tdata;  // Store the first
                            integer
45                      state <= ADD;
46                  end
47              end
48              ADD: begin
49                  if (s_axis_tvalid && s_axis_tlast) begin
50                      int_b <= s_axis_tdata;  // Store the second
                            integer
51                      sum <= int_a + s_axis_tdata;  // Perform the
                            addition
52                      state <= SEND;
53                  end
54              end
55              SEND: begin
56                  if (m_axis_tready) begin
57                      m_axis_tvalid <= 1'b1;
58                      m_axis_tdata <= sum;  // Send the sum to DMA
59                      m_axis_tlast <= 1'b1;  // Indicate the last
                            transfer
60                      state <= IDLE;  // Return to idle state
61                  end
62              end
63          endcase
64      end
65  end
66  endmodule
```

Code Explanation:

- **Parameters and Ports:** The module is parameterized to support different data widths (default is 32 bits). It has AXI-Stream input (`s_axis_tdata`, `s_axis_tvalid`, `s_axis_tready`, `s_axis_tlast`) and output (`m_axis_tdata`, `m_axis_tvalid`, `m_axis_tready`, `m_axis_tlast`) ports. The clock and reset signals (`aclk` and `aresetn`) drive the internal logic.
- **State Machine:** A simple state machine (`IDLE`, `ADD`, `SEND`) controls the flow of data.
  - **IDLE**: The module waits for valid data from the DMA on the AXI-Stream slave interface. When `s_axis_tvalid` is asserted, it captures the first operand (`int_a`).
  - **ADD**: When the second operand is received with `s_axis_tlast` asserted, it captures the second operand (`int_b`) and performs the addition (`sum = int_a + s_axis_tdata`).
  - **SEND**: Once the result is ready and the master (`m_axis_tready`) is ready to receive data, the module sends the sum on the AXI-Stream output, and

the `m_axis_tlast` signal indicates that this is the last data word in the transaction.

- **Handling Data Transfers:** The module uses the `s_axis_tready` signal to indicate it is ready to accept data and the `m_axis_tvalid` signal to indicate valid output data. These signals manage data flow between the DMA and the IP.

6.1.5. Jupyter Notebook for Testing AXI DMA Adder

The following Jupyter notebook code demonstrates how to use the AXI DMA interface to transfer data between system memory and the custom adder IP.

```
from pynq import Overlay, allocate

# Load the bitstream
overlay = Overlay("adder.bit")
dma = overlay.axi_dma_0  # AXI DMA instance from the block design

# Allocate memory buffers
input_buffer = allocate(shape=(2,), dtype='int32')  # Buffer for input integers
output_buffer = allocate(shape=(1,), dtype='int32')  # Buffer for the sum result

# Assign two integers to the input buffer
input_buffer[0] = 10
input_buffer[1] = 50

# Send the data using DMA (to your custom IP)
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)

# Wait for the transfer to complete
dma.sendchannel.wait()
dma.recvchannel.wait()

# Check the output
print(f"Sent integers: {input_buffer[:]}")
print(f"Received sum: {output_buffer[0]}")
```

```
Sent integers: [10 50]
Received sum: 60
```

Figure 6.4: Jupyter Notebook Code for Testing AXI DMA with Custom Adder

Code Breakdown:

1. **Loading the Bitstream:**

   ```
   overlay = Overlay('adder.bit')
   dma = overlay.axi_dma_0
   ```

   This loads the bitstream for the custom adder and retrieves the AXI DMA instance from the design.

2. **Allocating Buffers:**

   ```
   input_buffer = allocate(shape=(2,), dtype='int32')
   output_buffer = allocate(shape=(1,), dtype='int32')
   ```

   Two buffers are allocated: one for input data (two integers to be added) and one for the result (the sum).

3. **Transferring Data using DMA:**

```
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
```

The input data is transferred to the custom adder, and the result is transferred back to the output buffer.

4. **Waiting for Transfer Completion:**

```
dma.sendchannel.wait()
dma.recvchannel.wait()
```

These lines wait for both the send and receive transfers to complete.

5. **Checking the Output:**

```
print(f"Received sum: {output_buffer[0]}")
```

This prints the result of the addition operation (the sum of the two input integers).

6.2. Implementing Adder using AXI DMA on PYNQ-Z2

In this section, we will describe the Verilog code for a custom adder IP that sums multiple numbers coming through a data stream. The adder receives the data as a stream using AXI DMA and outputs the sum back as a stream.

6.2.1. Verilog Code for Summing Streamed Numbers

Below is the Verilog code for the custom AXI-Stream-based adder that sums streamed numbers.

Listing 6.2: Verilog Code for AXI DMA-based Streaming Adder

```
module adder(
    input wire          clk,
    input wire          resetn,

    // AXI Stream Slave Interface (Input from DMA)
    input wire [31:0]   s_axis_tdata,
    input wire          s_axis_tvalid,
    output wire         s_axis_tready,
    input wire          s_axis_tlast,

    // AXI Stream Master Interface (Output to DMA)
    output wire [31:0]  m_axis_tdata,
    output wire         m_axis_tvalid,
    input wire          m_axis_tready,
    output wire         m_axis_tlast
```

```
16      );
17
18      reg [31:0] sum;
19      reg valid;
20      reg ready;
21      reg last;
22
23      always @(posedge clk)
24      begin
25          if(!resetn)
26          begin
27              sum <= 'd0;              // Reset sum to zero
28              valid <= 'd0;            // Reset valid flag
29              last <= 'd0;             // Reset last flag
30              ready <= 'd1;            // Set ready to receive new
                    data
31          end
32          else begin
33              if (s_axis_tvalid && ready) begin
34                  sum <= sum + s_axis_tdata;  // Add streamed input
                        data to sum
35
36                  if (s_axis_tlast) begin
37                      valid <= 1'b1;       // Set valid signal when
                            sum is ready
38                      last <= 1'b1;        // Set last signal for
                            final output
39                      ready <= 1'b0;       // No longer ready to
                            receive until reset
40                  end
41              end
42
43              if (valid && m_axis_tready) begin
44                  valid <= 1'b0;           // Clear valid signal after
                        output is accepted
45                  last <= 1'b0;            // Clear last signal after
                        output is accepted
46              end
47          end
48      end
49
50      assign s_axis_tready = ready;
51      assign m_axis_tdata  = sum;
52      assign m_axis_tvalid = valid;
53      assign m_axis_tlast  = last;
54
55  endmodule
```

Code Explanation:

- **AXI Stream Slave Interface:** The input interface (`s_axis_tdata`, `s_axis_tvalid`, `s_axis_tready`, and `s_axis_tlast`) receives streamed data from the AXI DMA. The `s_axis_tdata` signal carries the input number, while `s_axis_tvalid` indicates that the data is valid. The `s_axis_tready` signal is used to inform the sender that the IP is ready to accept data.
- **AXI Stream Master Interface:** The output interface (`m_axis_tdata`, `m_axis_tvalid`, `m_axis_tready`, `m_axis_tlast`) sends the result (sum) back to the AXI DMA.

When the sum is ready, `m_axis_tvalid` is asserted, and the data is sent on `m_axis_tdata`. The `m_axis_tlast` indicates the end of the transaction.

- **Internal Logic:**
  - The `sum` register accumulates the sum of incoming data. It is cleared upon reset.
  - The state machine manages the data flow, controlling when the adder is ready to accept new data (`s_axis_tready`), and when the result is valid (`m_axis_tvalid`).
  - The adder processes data when `s_axis_tvalid` is asserted. The sum is computed, and the result is sent when `m_axis_tready` is asserted by the receiver (DMA).

6.2.2. Jupyter Code to Measure FPGA Performance Gain

The following Python code, using PYNQ, measures the performance of summing 10,000 random numbers using FPGA compared to CPU.

Listing 6.3: Jupyter Code for Measuring FPGA Performance Gain

```
# Import necessary libraries from PYNQ framework
import pynq
from pynq import Overlay
from pynq.lib.dma import DMA
import numpy as np
from pynq import allocate
import time

# Load the hardware overlay (bitstream) that contains the design with DMA and custom adder IP
overlay = Overlay("dma_adder.bit")
dma = overlay.axi_dma_0

# Generate an array of 1,000,000 random integers between 1 and 100 to be summed
random_numbers = np.random.randint(1, 100, size=1000000)

# Assign the generated random numbers to variable 't'
t = random_numbers

# Number of elements in the array
n = len(t)

# Initialize variables to store sum, software execution time, and hardware execution time
sum_cpu = 0
sw_exe_time = 0
hw_exe_time = 0

# -------- CPU (Software) Execution ---------
# Measure the execution time of summing the array using the CPU
start_time = time.time()
for i in range(n):
    sum_cpu = sum_cpu + t[i]  # Accumulate the sum of all numbers
stop_time = time.time()
```

```
33
34 # Calculate the CPU execution time for summing 1,000,000 numbers
35 sw_exe_time = stop_time - start_time
36
37 # Convert the array to a numpy array with int32 data type
38 array_t = np.array(t)
39 sample = array_t.astype(np.int32)
40
41 # -------- FPGA (Hardware) Execution ---------
42 # Allocate memory buffers for the input data (to FPGA) and output
       data (from FPGA)
43 with allocate(shape=(n,), dtype=np.int32) as in_buffer, \
44      allocate(shape=(1,), dtype=np.int32) as out_buffer:
45
46     # Copy the input data into the allocated input buffer
47     np.copyto(in_buffer, sample)
48
49     # Start measuring the FPGA execution time
50     start_time = time.time()
51
52     # Transfer the input buffer to the FPGA via DMA
53     dma.sendchannel.transfer(in_buffer)
54     # Transfer the output buffer to receive the result from FPGA
55     dma.recvchannel.transfer(out_buffer)
56
57     # Wait for both DMA send and receive channels to complete the
           transfers
58     dma.sendchannel.wait()
59     dma.recvchannel.wait()
60
61     # Measure the FPGA execution time
62     stop_time = time.time()
63     hw_exe_time = stop_time - start_time
64
65     # Print the output result (sum) from the FPGA
66     print("FPGA Result (Sum):", out_buffer[0])
67
68     # Close the buffers after use
69     in_buffer.close()
70     out_buffer.close()
71
72 # Print the execution times and performance comparison
73 print("FPGA (sum of 10,000 numbers) Execution Time: ", hw_exe_time,
       "seconds")
74 print("CPU Result (Sum):", sum_cpu)
75 print("CPU (sum of 10,000 numbers) Execution Time: ", sw_exe_time,
       "seconds")
76 print("Performance Gain: FPGA is", sw_exe_time / hw_exe_time, "
       times faster than CPU")
```

OUTPUT :

```
FPGA Result (Sum): 50063057
FPGA (sum of 10,000 numbers) Execution Time:  0.010853290557861328 seconds
CPU Result (Sum): 50063057
CPU (sum of 10,000 numbers) Execution Time:  4.0045270919799805 seconds
Performance Gain: FPGA is 368.9689380958657 times faster than CPU
```

Figure 6.5: Output of CPU and FPGA after summing 10,000 numbers. **NOTE**:This cpu results are obtained after runnig it on the PS (800MHz)of PYNQ-Z2 board itself and in a normal traditional CPU the time taken might be even lower.

Code Explanation:

- **FPGA Execution:** The `dma.sendchannel.transfer(in_buffer)` sends the data to the FPGA, and `dma.recvchannel.transfer(out_buffer)` retrieves the result. Both transfers are followed by waiting for the DMA channels to complete (`dma.sendchannel.wait()`, `dma.recvchannel.wait()`).
- **Performance Measurement:** The code compares the CPU and FPGA execution times. It prints the sum result from both the CPU and FPGA, as well as the time taken by each. The final output shows how many times faster the FPGA is compared to the CPU for summing 1,000,000 numbers.

Performance Comparison: FPGA vs CPU

The following figure presents a comparison between the execution time of summing 10,000 numbers using the FPGA and the CPU. The results demonstrate a significant performance gain when utilizing the FPGA for this task.

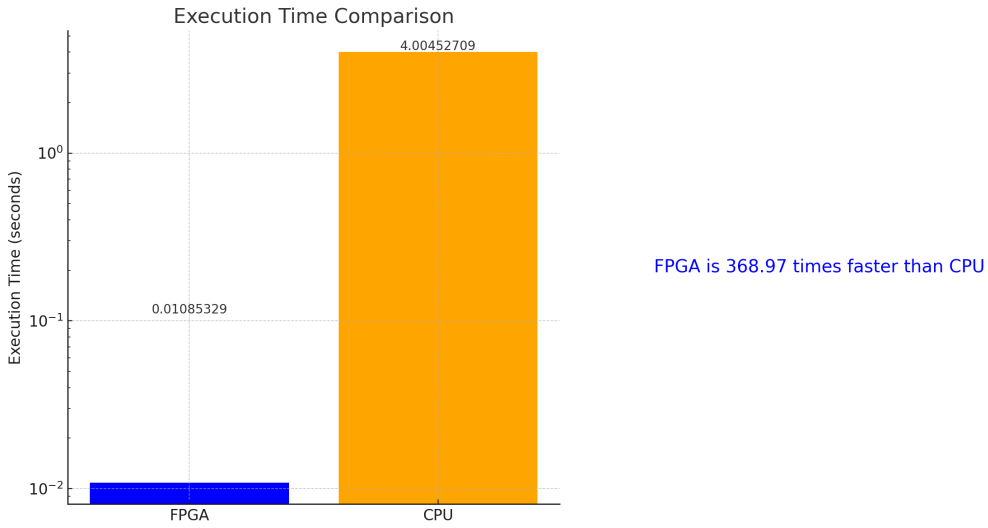

Figure 6.6: Performance comparison of CPU vs FPGA for summing 10,000 numbers

Analysis:
As shown in Figure 6.6, the FPGA is approximately ***369 times*** faster than the CPU when summing 10,000 random numbers. The results also indicate that both

the FPGA and the CPU produced the same sum, verifying the correctness of the FPGA operation.

Chapter 7

FIR Filter

7.1. Introduction to FIR Filter

A Finite Impulse Response (FIR) filter is a digital filter that processes input signals using a set of pre-defined filter coefficients. The output is the sum of the products of the filter coefficients and past and current input samples. FIR filters are popular because they are inherently stable, have a linear phase, and do not require feedback loops.

In this tutorial, we will compare the performance of an FIR filter implemented on a traditional CPU and an FPGA (using the PYNQ-Z2 platform). The comparison highlights the computational efficiency and speedup achievable using hardware acceleration.

7.2. FIR Theory and Design Choices

FIR filters are based on the convolution operation, where an output signal is computed as the weighted sum of current and past input signal values. The weights are the filter coefficients, which determine the frequency response of the filter.

$$y[n] = \sum_{k=0}^{M-1} h[k] \cdot x[n-k]$$

Where:

- $x[n]$ is the input signal at time step n,
- $h[k]$ are the filter coefficients,
- M is the number of taps (filter length).

7.2.1. Linear Phase

FIR filters are well-known for their ability to provide linear phase responses. This means that all frequency components of the input signal are delayed by the same

amount, preserving the wave shape of signals within the filter's passband. This is crucial for applications like image and audio processing, where phase distortion is undesirable.

7.2.2. Filter Design Choices

The design of an FIR filter involves the following considerations:

- **Filter Order:** The number of taps (coefficients) in the filter determines the filter order. A higher order filter provides a sharper frequency response but requires more computational resources.
- **Passband and Stopband:** FIR filters are typically designed to have a specific passband (the range of frequencies allowed through) and stopband (the range of frequencies that are attenuated).
- **Windowing Techniques:** Several windowing techniques, such as Hamming, Blackman, or Kaiser windows, are used to design FIR filters. The choice of window impacts the trade-off between the main lobe width and the side-lobe level in the frequency response.

The filter coefficients used in this document are based on a low-pass filter design using a Kaiser window, which balances the passband ripple and stopband attenuation. The order of the filter was selected based on the desired cutoff frequency and the level of attenuation required in the stopband.

7.3. FIR Filter Implementation on CPU

Below is the Python code used to implement the FIR filter on a CPU. The code uses the `scipy.signal.lfilter` function, which applies FIR coefficients to a generated signal.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import time

# FIR coefficients
coeffs = [-255, -260, -312, -288, -144, 153, 616, 1233, 1963, 2739,
          3474, 4081, 4481, 4620, 4481, 4081, 3474, 2739, 1963,
          1233, 616, 153, -144, -288, -312, -260, -255]

# Signal generation
T = 0.002
fs = 100e6  # 100 MHz sampling frequency
n = int(T * fs)
t = np.linspace(0, T, n, endpoint=False)

samples = 10000 * np.sin(0.2e6 * 2 * np.pi * t) + \
          1500 * np.cos(46e6 * 2 * np.pi * t) + \
          2000 * np.sin(12e6 * 2 * np.pi * t)

```

```
21 samples = samples.astype(np.int32)
22
23 # Measure CPU execution time
24 start_time = time.time()
25 sw_fir_output = lfilter(coeffs, 70e3, samples)
26 stop_time = time.time()
27
28 sw_exec_time = stop_time - start_time
29 print('CPU FIR execution time: ', sw_exec_time)
30
31 # Plot the result
32 plt.plot(t[:1000], samples[:1000], 'y-', label='Input signal')
33 plt.plot(t[:1000], sw_fir_output[:1000], 'g-', linewidth=2, label='
       FIR output')
34 plt.legend()
35 plt.show()
```

Explanation of CPU FIR Implementation:

- **Coefficients:** The FIR filter coefficients are defined as a list of integers. These coefficients determine the frequency response of the FIR filter.
- **Signal Generation:** The input signal is a combination of sinusoidal waves at different frequencies (0.2 MHz, 12 MHz, and 46 MHz). The signal is generated with a sampling frequency of 100 MHz and has a duration of 2 milliseconds.
- **Filtering:** The `lfilter()` function from the `scipy.signal` module applies the FIR filter to the input signal. This function convolves the input signal with the filter coefficients.
- **Execution Time Measurement:** The time taken to apply the FIR filter on the CPU is measured using the `time.time()` function.
- **Plotting:** The original and filtered signals are plotted to visualize the effect of the FIR filter on the input signal. The `matplotlib` library is used for plotting.

CPU FIR Output:

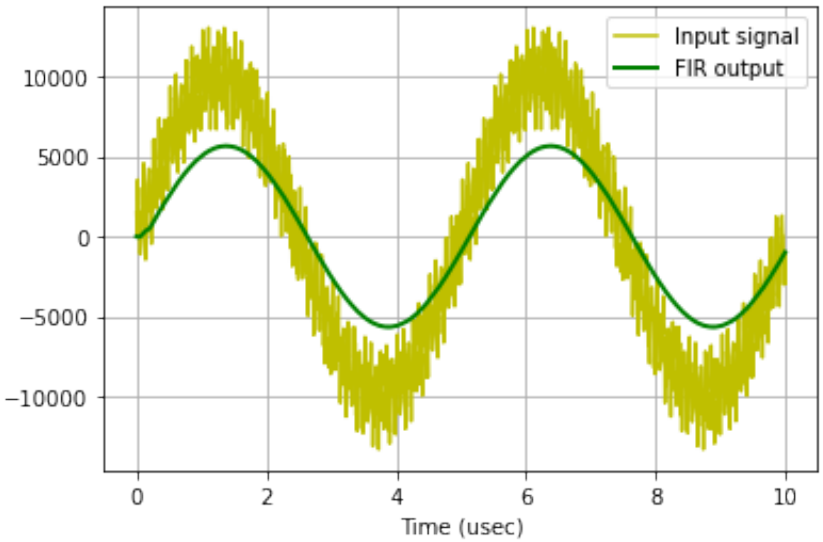

Figure 7.1: CPU FIR Output

CPU Execution Time: The CPU took approximately 0.08 seconds to apply the FIR filter to the generated signal.

7.4. FIR Filter Implementation on FPGA

The following block design shows the architecture used for implementing the FIR filter on the FPGA. The design uses the FIR Compiler IP, which applies the filter coefficients to the input signal. The design also incorporates DMA for high-speed data transfer between the Processing System (PS) and the FIR filter IP on the programmable logic (PL).

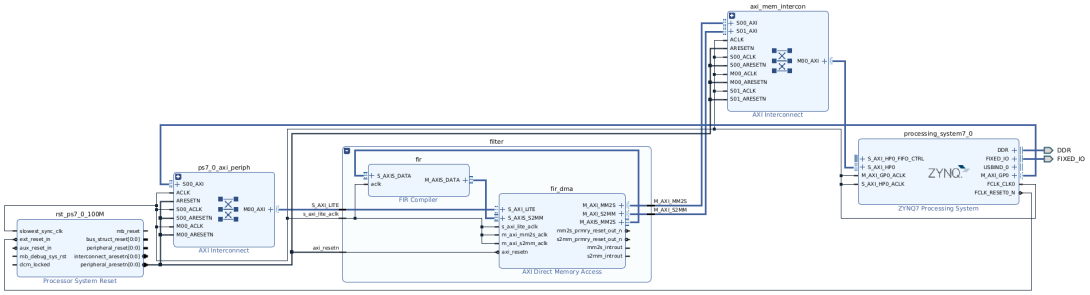

Figure 7.2: FIR Filter Block Design on FPGA using PYNQ-Z2

Block Design Description:

- **Zynq Processing System (PS):** The PS (ARM Cortex-A9) manages the data flow and communication between the user application and the FIR filter IP. It initiates the data transfer via AXI DMA.
- **AXI DMA:** The DMA is used to transfer the input signal from the PS to the FIR filter IP in the PL. Once the filter is applied, the DMA retrieves the filtered signal back to the PS.
- **FIR Compiler IP:** This block applies the FIR filter coefficients to the input signal in the hardware. It operates in a streaming mode, processing the input data as it arrives.
- **AXI Interconnect:** The AXI interconnect handles data flow between the PS and the FIR filter IP via DMA.

7.5. FIR Filter Configuration on FPGA

The FIR filter IP is configured using Vivado's IP customization window. Below are the screenshots of the settings used in the FIR IP.

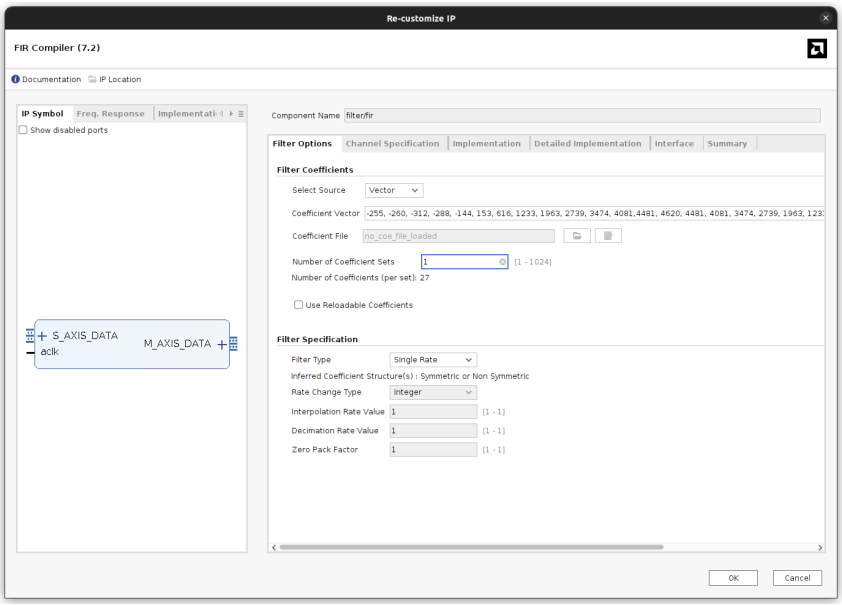

Figure 7.3: FIR Compiler Configuration: Coefficient Options

Explanation of Settings:

- **Coefficient Type:** Signed integer coefficients were selected. These coefficients define how the filter shapes the input signal by attenuating or amplifying certain frequencies.
- **Quantization:** Integer quantization was used for the coefficients to ensure compatibility with the input signal data format.
- **Input Data Type:** The input data is signed 32-bit integers, matching the bit-width of the input signal.
- **Filter Structure:** The filter is configured to use an inferred coefficient structure with symmetric coefficients to reduce the number of multiplications needed.

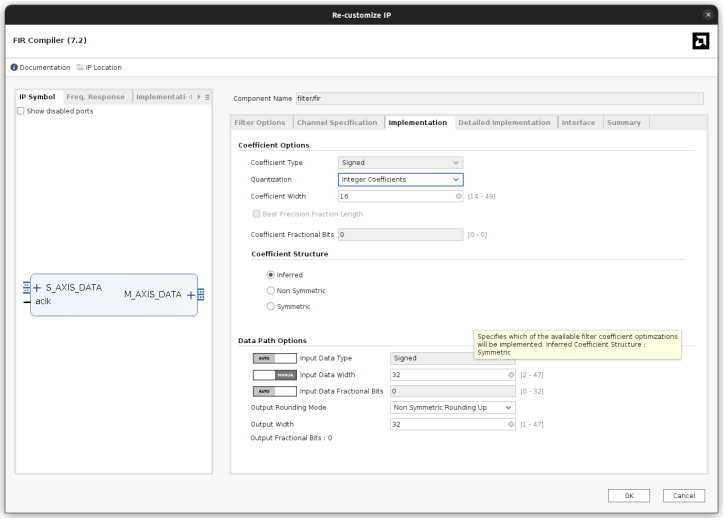

Figure 7.4: FIR Compiler Configuration: Filter Coefficients

Filter Coefficients:

– The filter is configured with 27 coefficients. The coefficients shape the filter's frequency response, determining which frequency components of the input signal are attenuated or passed through.

7.6. Jupyter Notebook for FPGA FIR Filter

The following Python code is executed on the PYNQ-Z2 board to apply the FIR filter using FPGA hardware. It demonstrates how data is sent from the PS to the FIR IP in the PL using DMA, and the filtered data is retrieved back into the PS for visualization.

```
import numpy as np
import matplotlib.pyplot as plt
from pynq import Overlay
import pynq.lib.dma
from pynq import allocate
import time

# Signal generation (same as in CPU implementation)
T = 0.002
fs = 100e6  # 100 MHz sampling frequency
n = int(T * fs)
t = np.linspace(0, T, n, endpoint=False)

samples = 10000 * np.sin(0.2e6 * 2 * np.pi * t) + \
          1500 * np.cos(46e6 * 2 * np.pi * t) + \
          2000 * np.sin(12e6 * 2 * np.pi * t)

samples = samples.astype(np.int32)

# Load the FPGA overlay
overlay = Overlay('fir.bit')
dma = overlay.filter.fir_dma

# Allocate buffers for input and output signals
with allocate(shape=(n,), dtype=np.int32) as in_buffer, \
     allocate(shape=(n,), dtype=np.int32) as out_buffer:

    # Copy the samples to the input buffer
    np.copyto(in_buffer, samples)

    # Trigger the DMA transfer and wait for the result
    start_time = time.time()
    dma.sendchannel.transfer(in_buffer)
    dma.recvchannel.transfer(out_buffer)
    dma.sendchannel.wait()
    dma.recvchannel.wait()
    stop_time = time.time()

    hw_exec_time = stop_time - start_time
    print('Hardware FIR execution time: ', hw_exec_time)

    # Plot the result
    plt.plot(t[:1000], samples[:1000], 'y-', label='Input signal')
```

```
44      plt.plot(t[:1000], out_buffer[:1000], 'g-', linewidth=2, label=
            'FIR output')
45      plt.legend()
46      plt.show()
```

Explanation of FPGA FIR Implementation:

- The FPGA bitstream for the FIR filter is loaded using the PYNQ overlay.
- The input signal is copied into a DMA buffer, which transfers the signal to the FIR filter IP in the PL.
- The DMA handles both sending the input signal to the FIR IP and receiving the filtered output.
- The execution time for the hardware FIR filter is measured and printed.
- The filtered output is plotted using `matplotlib`, similar to the CPU implementation.

FPGA FIR Output:

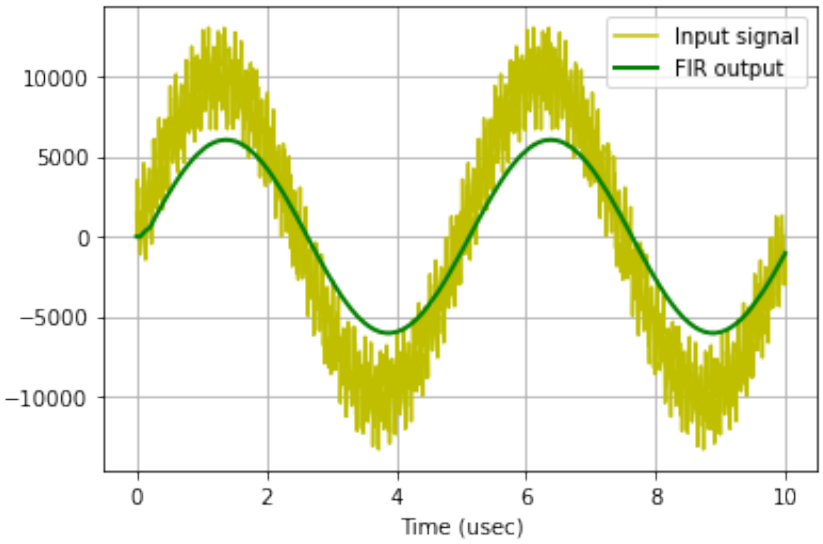

Figure 7.5: FPGA FIR Output

7.7. Performance Comparison: CPU vs FPGA

The execution time for the FIR filter on the CPU was approximately **0.08 seconds**, while the FPGA implementation took around **0.0029 seconds**. This results in a significant performance gain of **27.94x** when using the FPGA.

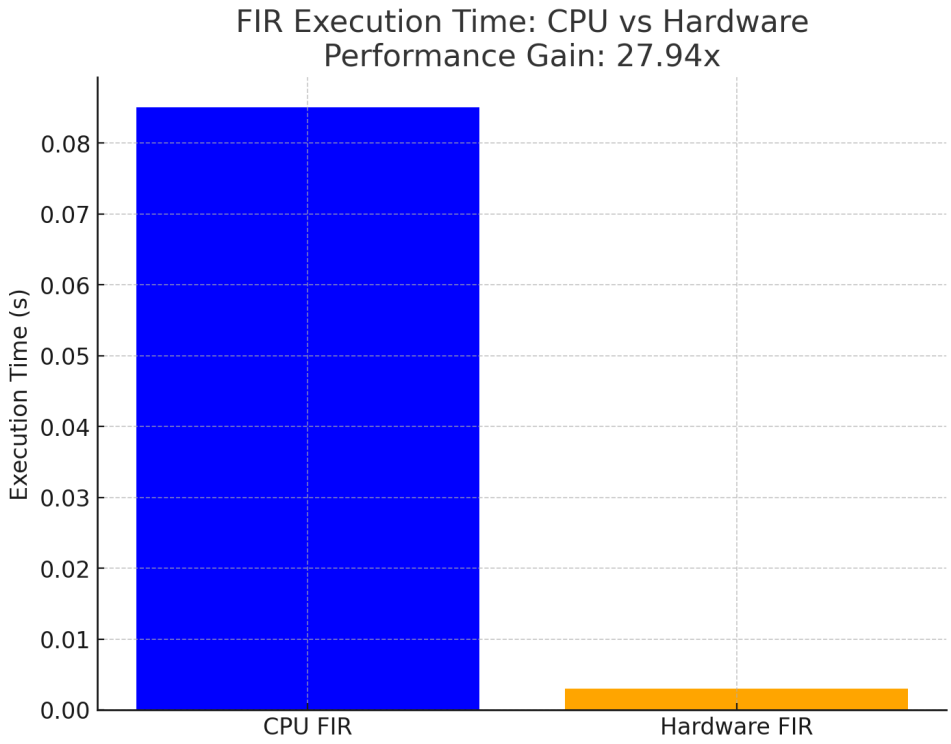

Figure 7.6: Performance Comparison of CPU vs FPGA FIR Execution Time

Conclusion: The FPGA-based FIR filter significantly outperforms the CPU-based implementation, achieving nearly 28x improvement in execution time. This demonstrates the advantage of hardware acceleration for computationally intensive signal processing tasks.

7.8. Resource Utilization on FPGA

In addition to performance benefits, implementing the FIR filter on an FPGA utilizes a variety of FPGA resources. This section outlines the main resources used by the FIR filter IP.

7.8.1. DSP Slices

DSP slices are specialized blocks within the FPGA designed to perform high-speed arithmetic operations such as multiplications and additions. FIR filters, which require multiple ***multiply-accumulate (MAC) operations***, rely heavily on these DSP slices to accelerate the computation of each multiplication between filter coefficients and input samples. By using DSP slices, the FPGA can achieve much higher throughput compared to a CPU that uses general-purpose processors.

7.8.2. LUTs and Flip-Flops

Look-Up Tables (LUTs) and **Flip-Flops** are the basic building blocks of FPGA logic. LUTs are used to implement combinatorial logic, while flip-flops are used for storing state information. In the FIR filter implementation, LUTs are used for address generation and control logic, while flip-flops are used to store intermediate values between stages of the filter pipeline.

7.8.3. Block RAM (BRAM)

The FIR filter stores filter coefficients and intermediate results in **Block RAM (BRAM)**. The BRAM serves as fast, on-chip memory that is directly accessible by the FIR IP and the DMA engine, allowing for quick access to filter coefficients without needing to fetch them from external memory.

7.8.4. Summary of Resource Usage

- **DSP Slices:** FIR filters heavily rely on DSP slices for multiplication. The more taps a filter has, the more DSP slices are consumed.
- **LUTs:** LUTs are used for control and data routing logic, including managing the data flow between the DMA and FIR IP.
- **Flip-Flops:** Flip-flops are utilized in storing the intermediate results of filtering stages, allowing the pipeline to function efficiently.
- **BRAM:** Used to store the coefficients and intermediate data. FIR filters with a large number of taps or high throughput requirements may require more BRAM.

7.9. Conclusion

In conclusion, this comparison demonstrated the significant performance gains that can be achieved by implementing an FIR filter on an FPGA versus a CPU. The FPGA provided a speedup of nearly 28x, demonstrating the power of hardware acceleration for signal processing tasks.

- **Real-Time Processing:** Implement the FIR filter for real-time data streaming applications, such as live audio or video filtering.
- **Adaptive Filtering:** Explore more complex filter designs, such as adaptive FIR filters, which can dynamically adjust coefficients based on the input signal.
- **Multi-FPGA Systems:** Investigate the potential for using multiple FPGAs in parallel to further accelerate complex filtering tasks.

Chapter 8

Image Dimming using AXI DMA

8.1. Image Dimming using AXI DMA on PYNQ-Z2

This tutorial describes the implementation of image dimming using AXI DMA on the PYNQ-Z2 board. The goal is to receive image data via the PS (Processing System), send it to the PL (Programmable Logic) for processing (dimming), and return the processed image back to the PS via the AXI DMA interface. The PS will then send the dimmed image back to the host system, which initially sent the images for processing. The key operation performed on each image is a pixel-wise dimming operation, which reduces the intensity of each pixel by half, resulting in a darker version of the image.

8.1.1. Image Dimming Explanation

Image dimming is a process where the intensity of each pixel in an image is reduced to create a darker version of the original image. This is commonly used in applications where brightness adjustment is required, or when an image is meant to convey a different mood.

In this example, the dimming operation was applied to an image. The pixel-wise operation reduces the intensity of each pixel by half, resulting in a dimmer version of the image. Below are the original and dimmed versions of the image for comparison.

Original Image:

Figure 8.1: Original Image

Dimmed Image:

Figure 8.2: Dimmed Image after applying the Dimming Operation

Explanation of the Dimming Process:

- **Pixel-wise Dimming:** The custom IP reduces the intensity of each pixel by half by applying a **bitwise right-shift** operation. For example, a pixel with a value of 200 in the original image would be dimmed to 100.
- **Image Representation:** Each pixel's intensity is represented as an 8-bit value in one channel in an RGB image. By performing the dimming operation, the brightness of the image decreases uniformly across all pixels.
- **Visual Result:** The dimmed image appears darker, but the overall structure and detail of the image are preserved. This process ensures a smooth transition in pixel intensity while maintaining the image's quality.

In this case, the original and dimmed images showcase how the operation modifies the pixel intensity without altering the content of the image.

8.2. Block Design Description

The block design for this project implements communication between the PS and the PL using the AXI Direct Memory Access (DMA) interface. The custom IP (written in Verilog) applies the dimming operation to the image, and the processed image data is transferred back to the PS.

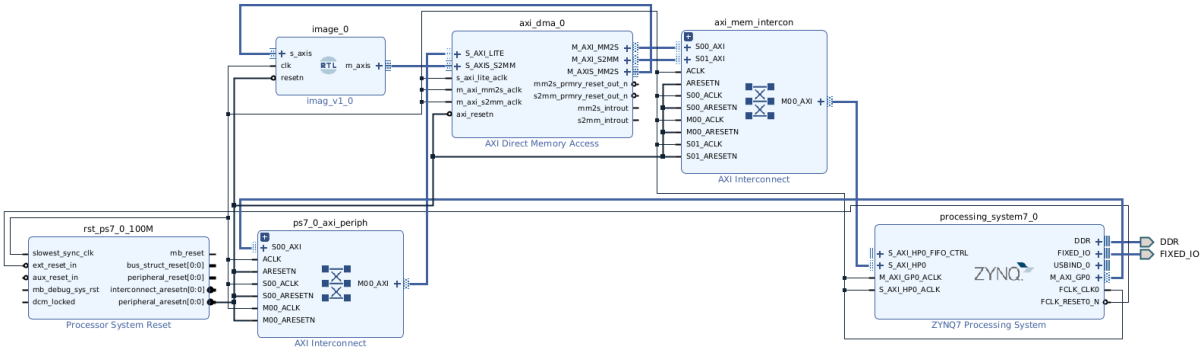

Figure 8.3: Block Design for Image Dimming using AXI DMA

Key Components in the Block Design:

- **Processing System (PS7):** This is the ARM Cortex-A9 processing subsystem of the Zynq chip on the PYNQ-Z2 board. The PS runs Linux and is responsible for managing socket communication to receive and send images from a host computer. Additionally, it controls the transfer of data (images) between the PS and the custom logic (PL) through the AXI DMA interface.
- **AXI DMA (Direct Memory Access):** The AXI DMA is crucial for transferring data between the PS and the PL efficiently. It handles data transfer without involving the CPU, thus speeding up operations. The DMA has two main channels:
  - **Memory Map to Stream (MM2S) Channel:** This channel transfers data from the PS's memory (DDR) to the PL as a stream. In our case, the image data is sent to the custom IP in the PL via this channel.
  - **Stream to Memory Map (S2MM) Channel:** This channel takes the processed data (dimmed image) from the PL and writes it back to the PS's memory for further processing or transmission.
- **Custom IP (Image Dimming):** This is the custom Verilog IP block that performs the image dimming operation. The IP receives the image data in 8-bit format (or a color channel of an RGB image) from the AXI DMA. The dimming operation reduces the intensity of each pixel by half by performing a right-shift operation on the pixel data. After processing, the dimmed image is sent back to the PS via the AXI DMA.

- **AXI Interconnect:** The AXI Interconnect handles the connections between the various AXI interfaces in the design. It links the AXI DMA, the custom IP, and the PS, ensuring the correct flow of data between the different components.
- **Reset and Clocking:** The design includes global reset and clock signals. These ensure that all components, including the PS, DMA, and custom IP, are synchronized and initialized properly. The reset block ensures the design starts in a known state, and the clocking block generates the clock signals required to drive the design.

8.3. Data Flow Overview

Step-by-Step Image Processing Flow:

1. The host computer sends a total of 50 images, one by one, to the PYNQ-Z2 board via socket programming.
2. The PS receives each image and uses the AXI DMA MM2S channel to transfer the image data to the custom IP block in the PL.
3. The custom IP processes the image, applying the dimming operation to each pixel in real-time.
4. The processed (dimmed) image is transferred back from the PL to the PS through the AXI DMA S2MM channel.
5. Once the PS receives the dimmed image, it sends it back to the host computer, which can compare the original and dimmed images to verify correctness and measure performance.

This design efficiently uses the AXI Stream protocol for high-speed data transfer between the PS and PL, leveraging the FPGA's hardware acceleration for image processing.

8.4. Verilog Code for Image Dimming Custom IP

The custom IP block implemented in Verilog is responsible for dimming the intensity of the pixels in the image by halving the intensity values. This is achieved by performing a right shift operation on the 8-bit pixel data, effectively dividing the pixel intensity by 2.

Listing 8.1: Verilog Code for Image Dimming

```
1 `timescale 1ns / 1ps
2
3 module image(
4     input wire          clk,
5     input wire          resetn,
6
7     // AXI Stream Slave Interface (Input from DMA)
8     input wire [7:0]    s_axis_tdata,
9     input wire          s_axis_tvalid,
```

```
    output wire          s_axis_tready,
    input wire           s_axis_tlast,

    // AXI Stream Master Interface (Output to DMA)
    output wire [7:0]    m_axis_tdata,
    output wire          m_axis_tvalid,
    input wire           m_axis_tready,
    output wire          m_axis_tlast
    );

    // Pass-through connections with image dimming
    assign s_axis_tready = m_axis_tready;
    assign m_axis_tdata  = s_axis_tdata >> 1;  // Dimming by right-
       shifting (divide by 2)
    assign m_axis_tvalid = s_axis_tvalid;
    assign m_axis_tlast  = s_axis_tlast;
endmodule
```

Code Explanation:

- **AXI Stream Slave Interface:** The input interface (`s_axis_tdata`, `s_axis_tvalid`, `s_axis_tready`, and `s_axis_tlast`) receives the image data as a stream from the AXI DMA MM2S channel. The pixel data is 8 bits wide, representing grayscale intensity (or one color channel of an RGB image).
- **AXI Stream Master Interface:** The output interface (`m_axis_tdata`, `m_axis_tvalid`, `m_axis_tready`, `m_axis_tlast`) sends the dimmed image data back to the AXI DMA S2MM channel. The dimmed image data will be stored in memory on the PS side.
- **Dimming Operation:** The line `assign m_axis_tdata = s_axis_tdata >> 1;` performs the dimming operation. This right-shift operation effectively halves the intensity of each pixel, making the image appear darker. For example, a pixel with a value of 200 would be dimmed to 100.
- **Pass-through Signals:** The `s_axis_tready`, `m_axis_tvalid`, and `m_axis_tlast` signals propagate the handshaking control signals required for the AXI Stream protocol to manage the flow of data between the PS and PL.

8.4.1. Socket Code for Sending and Receiving Images

The following Python code establishes a socket connection between a host computer and the PYNQ-Z2 board. It sends images from the host to the board, where the dimming operation is performed, and then receives the processed images back. The images are saved to a designated folder for further evaluation.

```
import socket
import os
import cv2
import numpy as np

# Directory containing the images to be sent
image_dir = "socket_images"
image_files = os.listdir(image_dir)

```

```
# Directory to save the processed images
output_dir = "Dimmed_Images"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# IP and port of the PYNQ board
pynq_ip = '10.21.239.174'  # Replace with the actual IP address of your PYNQ board
port = 9092

# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((pynq_ip, port))

for image_file in image_files[:50]:
    image_path = os.path.join(image_dir, image_file)

    # Read the image
    image = cv2.imread(image_path)
    if image is None:
        print(f"Error reading image: {image_path}")
        continue  # Skip this image and proceed with the next

    height, width, channels = image.shape
    image_info = f"{height},{width},{channels}".encode()

    # Send image dimensions first
    client_socket.sendall(image_info)

    # Send the image data
    image_bytes = image.flatten().tobytes()
    client_socket.sendall(image_bytes)

    # Receive the processed (dimmed) image back
    processed_image_bytes = b''
    expected_size = height * width * channels
    while len(processed_image_bytes) < expected_size:
        packet = client_socket.recv(expected_size - len(processed_image_bytes))
        if not packet:  # Break if no data is received
            break
        processed_image_bytes += packet

    # Reshape the image and save it in the new folder
    processed_image = np.frombuffer(processed_image_bytes, dtype=np.uint8).reshape((height, width, channels))
    output_image_path = os.path.join(output_dir, f"dimmed_{image_file}")
    cv2.imwrite(output_image_path, processed_image)

client_socket.close()
```

Explanation of Python Code:

- **Directory Setup:** The code begins by defining the directories for reading input images (‘image_dir‘) and saving the processed (dimmed) images (‘output_dir‘). If the output directory does not exist, it is created using `os.makedirs(output_dir)`.

- **Socket Setup:** A socket is created using `socket.socket()` for TCP communication. The host computer connects to the PYNQ-Z2 board using its IP address (`pynq_ip`) and port (`9092`). The `client_socket.connect()` function establishes the connection between the host and the PYNQ board.
- **Reading and Sending Images:** The code iterates over the images in the `image_dir` directory and reads each image using OpenCV (`cv2.imread()`). The dimensions of each image (height, width, and channels) are sent to the PYNQ board using `client_socket.sendall()`. Then, the image is flattened into a byte array (`image.flatten().tobytes()`) and sent to the board for processing.
- **Receiving Processed Images:** Once the dimming operation is performed on the PYNQ board, the processed image is sent back to the host. The code uses a loop to receive the image data in chunks until the entire image is received. The received bytes are reshaped into the original image format using `np.frombuffer()` and `np.reshape()`.
- **Saving Processed Images:** The processed (dimmed) images are saved to the `Dimmed_Images` directory with the prefix `dimmed_` added to the original image filenames.
- **Closing the Socket:** After processing all images, the socket connection is closed using `client_socket.close()`.

Key Points:

- The code efficiently sends and receives image data over a TCP socket using the PYNQ-Z2 board as the server.
- The dimming operation is performed in real-time, and the images are saved locally for further evaluation.
- This setup allows for high-speed data transfer between the host and the PYNQ board, leveraging the board's FPGA for image processing.

8.4.2. Transferring Files via SCP and Running Python Code via SSH

To deploy the bitstream and Python code to the PYNQ-Z2 board and execute the dimming operation, we use Secure Copy Protocol (SCP) to transfer the necessary files and Secure Shell (SSH) to run the code. Below are the steps involved.

Transferring Files via SCP

The following command shows how the required files (bitstream, hardware handoff file, and Python script) are copied from the local machine to the PYNQ-Z2 board using SCP.

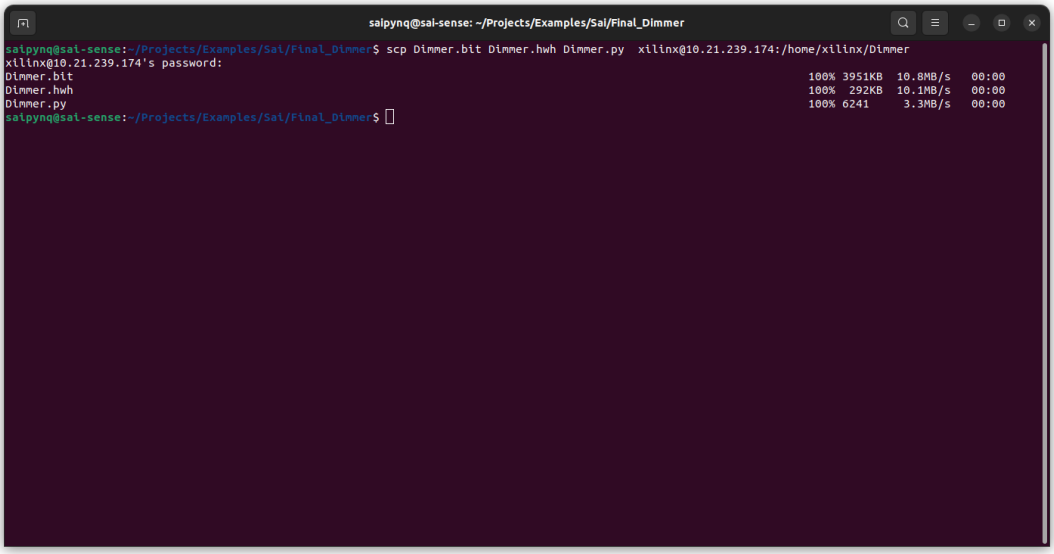

Figure 8.4: SCP Command Used to Transfer Files to the PYNQ-Z2 Board

In this screenshot, the files:

- `Dimmer.bit` – The bitstream file to configure the FPGA.
- `Dimmer.hwh` – The hardware handoff file for interfacing with the Python libraries.
- `Dimmer.py` – The Python script for running the image dimming operation.

are transferred to the `Dimmer` directory on the PYNQ-Z2 board.

The SCP command is structured as follows:

```
scp Dimmer.bit Dimmer.hwh Dimmer.py xilinx@<PYNQ-IP-Address>:/home/xilinx/Dimmer
```

Where:

- `xilinx@<PYNQ-IP-Address>` refers to the username and IP address of the PYNQ-Z2 board.
- `/home/xilinx/Dimmer` is the target directory on the PYNQ-Z2 where the files are to be transferred.

Running the Python Code via SSH

Once the files are transferred, we connect to the PYNQ-Z2 board using SSH to execute the Python script. The following screenshot shows how to connect to the board using SSH and navigate to the directory containing the transferred files.

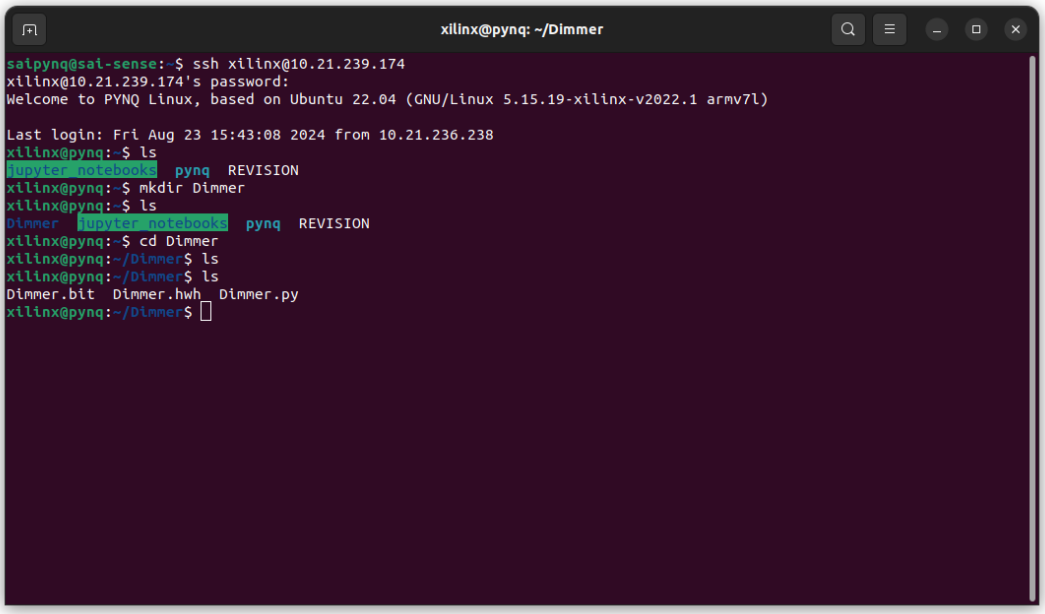

Figure 8.5: SSH Session on PYNQ-Z2 and Executing Python Code

- `ssh xilinx@<PYNQ-IP-Address>` connects to the PYNQ-Z2 board via SSH.
- Once inside the `Dimmer` directory, the `ls` command shows that the files `Dimmer.bit`, `Dimmer.hwh`, and `Dimmer.py` are successfully transferred.
- Finally, the Python script is executed using `sudo -E python3 Dimmer.py`.

By using SCP for file transfer and SSH for running Python code on the PYNQ-Z2 board, we can efficiently deploy and test our image processing applications on the FPGA.

8.5. Python Code for Image Dimming using AXI DMA

Below is the Python code that tests the functionality of image dimming using AXI DMA on the PYNQ-Z2 board.

```
import socket
import numpy as np
import cv2
from pynq import Overlay
from pynq.lib.dma import DMA
from pynq import allocate
import time  # For performance measurement

# Load the overlay and DMA
overlay = Overlay("Dimmer.bit")
dma = overlay.axi_dma_0

# Create a socket for the server
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
15 server_socket.bind(('0.0.0.0', 9092))  # Listen on all available
       interfaces
16 server_socket.listen(1)
17
18 print("Waiting for a connection...")
19 conn, addr = server_socket.accept()
20 print(f"Connected by {addr}")
21
22 # Initialize variables for performance measurement
23 num_images_processed = 0  # Count the number of images processed
24 total_fpga_processing_time = 0  # To sum up the FPGA processing
       times
25
26 # Start timer for total image processing
27 overall_start_time = time.time()  # Timer to measure total image
       processing time
28
29 while True:
30     # Receive image dimensions first
31     image_info = conn.recv(1024).decode()
32     if not image_info:
33         break  # Exit the loop if no data is received
34
35     height, width, channels = map(int, image_info.split(','))
36
37     try:
38         # Allocate buffers for input and output
39         input_buffer = allocate(shape=(height * width * channels,),
               dtype=np.uint8)
40         output_buffer = allocate(shape=(height * width * channels,)
               , dtype=np.uint8)
41     except RuntimeError as e:
42         print(f"Memory allocation failed: {e}")
43         continue  # Skip to the next iteration or handle
               accordingly
44
45     # Receive image bytes
46     image_bytes = b''
47     while len(image_bytes) < height * width * channels:
48         packet = conn.recv(4096)
49         if not packet:
50             break
51         image_bytes += packet
52
53     # Copy the image bytes to the input buffer
54     np.copyto(input_buffer, np.frombuffer(image_bytes, dtype=np.
           uint8))
55
56     # Process the image in chunks
57     rows_per_chunk = 8
58     for i in range(0, height, rows_per_chunk):
59         start_row = i
60         end_row = min(i + rows_per_chunk, height)
61
62         # Transfer current chunk
63         dma.sendchannel.transfer(input_buffer[start_row * width *
               channels:end_row * width * channels])
```

```
64          dma.recvchannel.transfer(output_buffer[start_row * width *
                channels:end_row * width * channels])
65
66          # Wait for transfer to complete
67          dma.sendchannel.wait()
68          dma.recvchannel.wait()
69
70      # Convert the output buffer to a numpy array for the processed
            image
71      processed_image = np.reshape(output_buffer[:height * width *
            channels], (height, width, channels))
72
73      # Send the processed image back to the host
74      conn.sendall(output_buffer[:height * width * channels])
75
76      # Increment image count
77      num_images_processed += 1
78
79  # End timer for total image processing
80  overall_end_time = time.time()
81  total_fpga_processing_time = overall_end_time - overall_start_time
82
83  # After all images are processed, calculate the throughput
84  if num_images_processed > 0:
85      throughput = num_images_processed / total_fpga_processing_time
              # Images per second
86      print(f"Total images processed: {num_images_processed}")
87      print(f"Total FPGA processing time: {total_fpga_processing_time
            :.6f} seconds")
88      print(f"Throughput: {throughput:.2f} images per second")
89  else:
90      print("No images were processed.")
91
92  # Close connections and destroy windows
93  conn.close()
94  server_socket.close()
```

Explanation of Python Code:

- **Overlay and DMA Setup:**
  - The bitstream `"Dimmer.bit"` is loaded onto the FPGA using the PYNQ 'Overlay' class, which configures the programmable logic.
  - The DMA object is set up by accessing `axi_dma_0`, which is responsible for transferring image data between the PS and PL without involving the CPU.
- **Socket Setup:**
  - A TCP socket is created to facilitate communication between the host and the PYNQ-Z2 board. The server listens on all network interfaces (0.0.0.0) and port 9092 for incoming connections.
  - After accepting a connection from the client, the server and client can exchange image data.
- **Buffer Allocation:**

    - Memory buffers are allocated using the PYNQ '**allocate**' function, which allocates memory in the PS. This allows efficient data transfer between the PS and PL.
    - Separate input and output buffers are allocated to store the original and processed images, respectively. The dimensions of the buffers match the size of the incoming image.
- **Receiving Image Data:**
    - The image dimensions (**height, width, and channels**) are first received from the client, followed by the image data itself.
    - The data is received in **chunks** of 4096 bytes using the 'recv()' function and stored in 'image_bytes' until the full image is received.
- **Image Processing in Chunks:**
    - To ensure efficient processing, the image is processed in smaller chunks of 8 rows at a time. This prevents overwhelming the FPGA's resources.
    - For each chunk, the input buffer is transferred to the FPGA using the DMA 'sendchannel.transfer()' function. After processing, the result is transferred back into the output buffer using the 'recvchannel.transfer()' function.
    - Both '**sendchannel.wait()**' and '**recvchannel.wait()**' ensure that the DMA transfers complete before proceeding to the next chunk.
- **Reshaping the Output:**
    - After processing, the output buffer is reshaped into a 3D numpy array that represents the processed image. The array is reshaped to match the original image dimensions (height, width, channels).
- **Sending Processed Image Back:**
    - The dimmed image is sent back to the host using the 'sendall()' function, sending the processed image data byte by byte.
- **Performance Measurement:**
    - The time taken to process all images is measured using 'time.time()' at the start and end of the image processing loop. This allows the calculation of total FPGA processing time.
- **Calculating Throughput:**
    - The throughput is calculated as the number of images processed divided by the total FPGA processing time, giving the number of images processed per second.
- **Closing Connections:**
    - Once all images are processed, the socket connection is closed, and the resources used by the server are released.

8.5.1. Output of the Image Dimming Code

The image dimming functionality was successfully executed, and the output of the Python code is shown in the figure below. This output showcases the performance

metrics of the system, including the processing time and throughput for both the Programmable Logic (PL) and the Processor (PS).

Output:

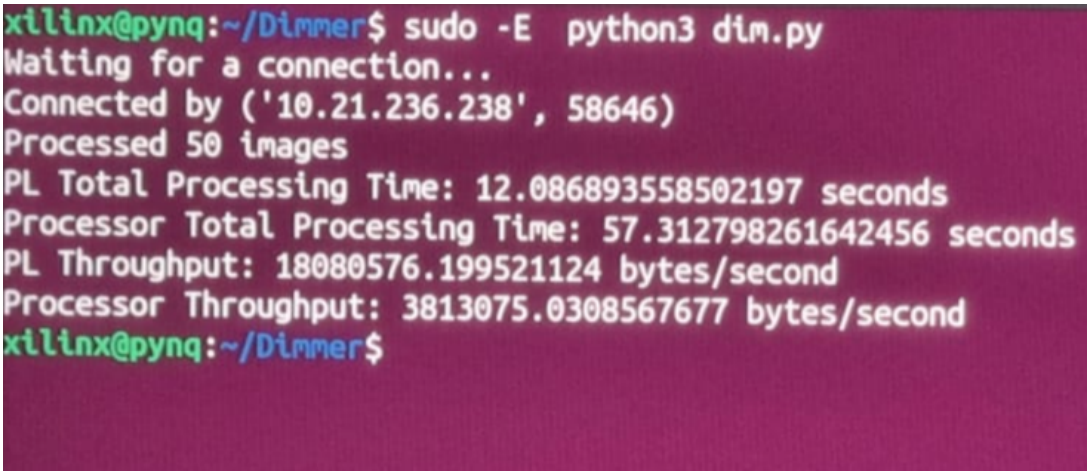

Figure 8.6: Output of the Image Dimming Python Code

Explanation of Results:

- **Processed 50 Images:** The system processed a total of **50** images sent from the host computer to the PYNQ-Z2 board. Each image underwent pixel-wise dimming and was sent back to the host.
- **PL Total Processing Time:** The Programmable Logic (FPGA) took approximately **12.086 seconds** to process the 50 images. This indicates the total time spent by the FPGA fabric to dim the images.
- **Processor Total Processing Time:** The ARM Cortex-A9 processor (PS) took around **57.313 seconds** for its part of the task. This includes overheads like socket communication, data transfers between the PS and the PL, and managing the DMA.
- **PL Throughput:** The PL achieved a throughput of approximately **18,080,576 bytes/second**, indicating the speed at which data is processed by the FPGA. This value represents the number of bytes processed per second by the PL when performing the dimming operation.
- **Processor Throughput:** The processor (PS) achieved a throughput of approximately **3,813,075 bytes/second**. This value is lower than the PL throughput, as the processor has additional responsibilities like managing the DMA, communication, and data handling.

Key Insights:

- The PL is significantly faster at processing the images than the processor, highlighting the advantages of using FPGA-based hardware acceleration for such tasks.

- The processing time of the PS includes the time for data transfers and communication, which adds to the overall execution time compared to the more focused processing done by the PL.
- The throughput values indicate the efficiency of data handling in the system. The PL's higher throughput suggests that FPGA-based operations can handle data-intensive tasks more efficiently than general-purpose processors.

This output confirms that the image dimming operation was successfully implemented, and the performance of the FPGA (PL) greatly surpasses that of the processor (PS) for this specific task.

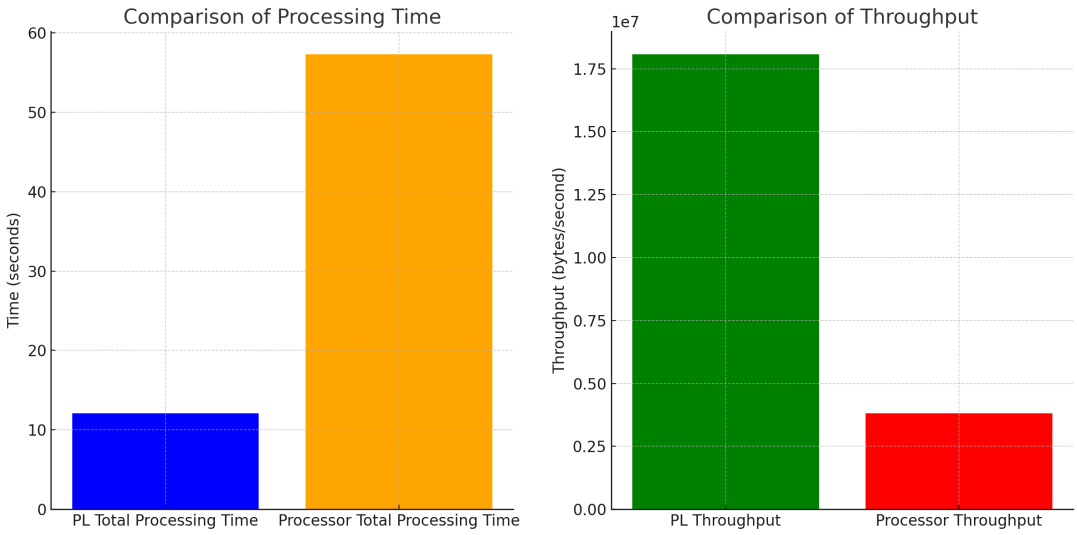

Figure 8.7: Comparision of image dimming on CPU vs FPGA

8.5.2. Conclusion

In this project, we successfully implemented an image dimming operation on the PYNQ-Z2 board by leveraging the AXI DMA for efficient data transfer between the Processing System (PS) and the Programmable Logic (PL). The FPGA's hardware acceleration enabled the rapid dimming of image pixels, demonstrating a significant improvement in processing speed compared to a purely software-based approach.

The use of the custom Verilog IP, along with AXI DMA for memory-mapped and streaming data, efficiently handled the pixel-wise operation. The image data was processed in chunks to manage memory effectively, while the PS handled socket communication with a host computer for sending and receiving images.

This project showcases how ***FPGAs can be utilized to offload computational tasks from the CPU, achieving real-time image processing with minimal latency***. The techniques applied here can be extended to more complex image processing tasks such as **edge detection, filtering, and real-time video processing**.

By combining **hardware acceleration with software control**, we demonstrated the power of the PYNQ-Z2 platform in a practical image processing application, making it a suitable choice for a wide range of **embedded system designs**.

Chapter 9

Running Neural Networks on FPGA

9.1. Implementing MNIST MLP on FPGA with ZyNet

In this section, we discuss the implementation of a neural network for MNIST digit classification on the PYNQ-Z2 board using the ZyNet architecture. This implementation leverages FPGA resources to accelerate inference of a Multilayer Perceptron (MLP) model. The ZyNet framework provides a structured approach for deploying neural networks on FPGA.

9.1.1. Overview of ZyNet

ZyNet is an FPGA-based neural network accelerator specifically optimized for low-power and real-time applications. It utilizes the ARM Cortex-A9 cores and the Programmable Logic (PL) available on the Zynq System on Chip (SoC) to achieve efficient parallel processing for neural network inference. The architecture leverages the AXI interface for communication between the Processing System (PS) and PL. ***ZyNet*** employs **quantization** and efficient memory usage to accelerate neural network operations without sacrificing accuracy significantly.

9.1.2. Block Diagram of MNIST MLP Implementation Using ZyNet

The block diagram of our MNIST MLP implementation is shown in Figure 9.1. This design utilizes the following components to perform neural network inference on FPGA:

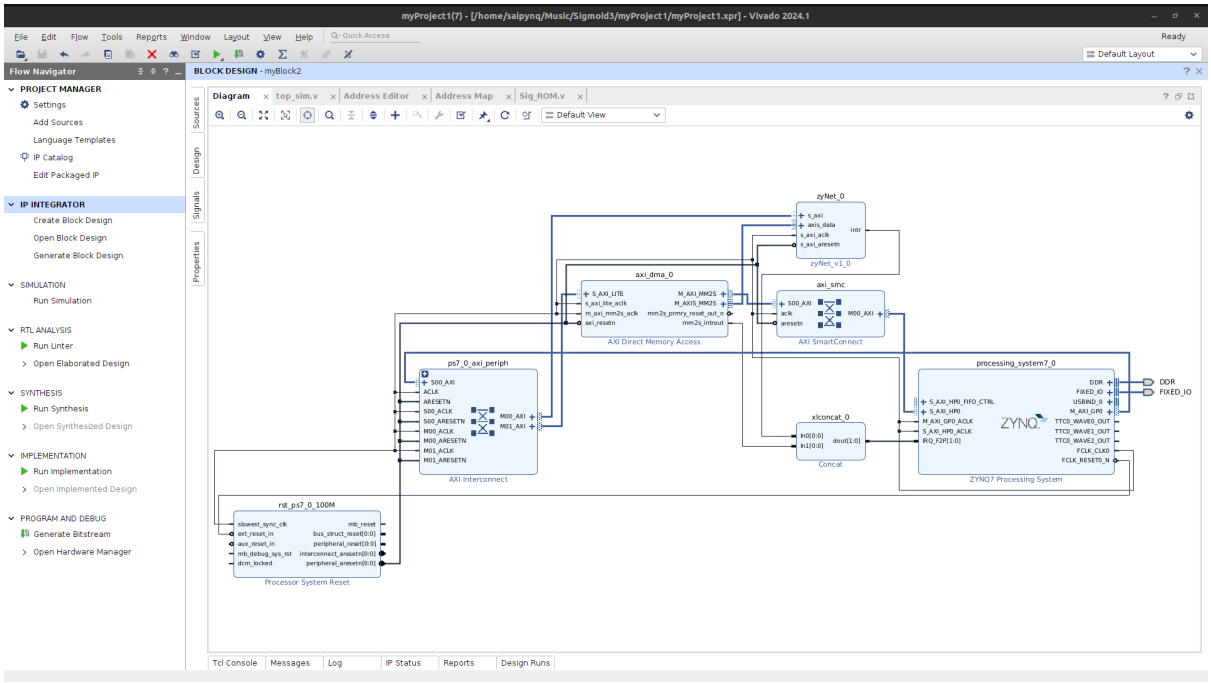

Figure 9.1: Block Design of MNIST MLP Implementation using ZyNet

Key Components of the Block Design:

1. **Processing System (PS):** The PS runs the application software, handles data pre-processing, and orchestrates the overall control of data transfer between the host and FPGA. In this setup, the PS loads the image data, transfers it to the FPGA for inference, and retrieves the result.
2. **AXI DMA Interface:** The AXI Direct Memory Access (DMA) module facilitates efficient data movement between the PS and the ZyNet module. By using AXI DMA, data can be transferred in a streaming fashion to the neural network engine on the programmable logic.
3. **ZyNet Accelerator Core:** This is the core neural network processing engine on the programmable logic. The ZyNet accelerator consists of layers and activation functions mapped directly to FPGA resources, enabling parallel execution of MLP operations. Each layer in the MLP network is implemented as a series of matrix multiplications, accumulations, and non-linear activations.
4. **AXI Interconnect and Control Logic:** The interconnect facilitates communication between different components within the programmable logic, while the control logic manages the data flow, synchronization, and coordination between layers.
5. **Concatenation Module (xlconcat):** This module consolidates interrupts from the ZyNet core, signaling the PS when the computation of an inference is complete and the result is ready for retrieval.

This design leverages the inherent parallelism of FPGA architecture to perform rapid matrix multiplications and activations, which are the core computations in an MLP.

9.1.3. Current Status: Simulation Output of MNIST MLP

As of now, the design is in the simulation phase, and we have successfully obtained output for the MNIST MLP model in a simulated environment. Figure 9.2 provides the output from the functional simulation run.

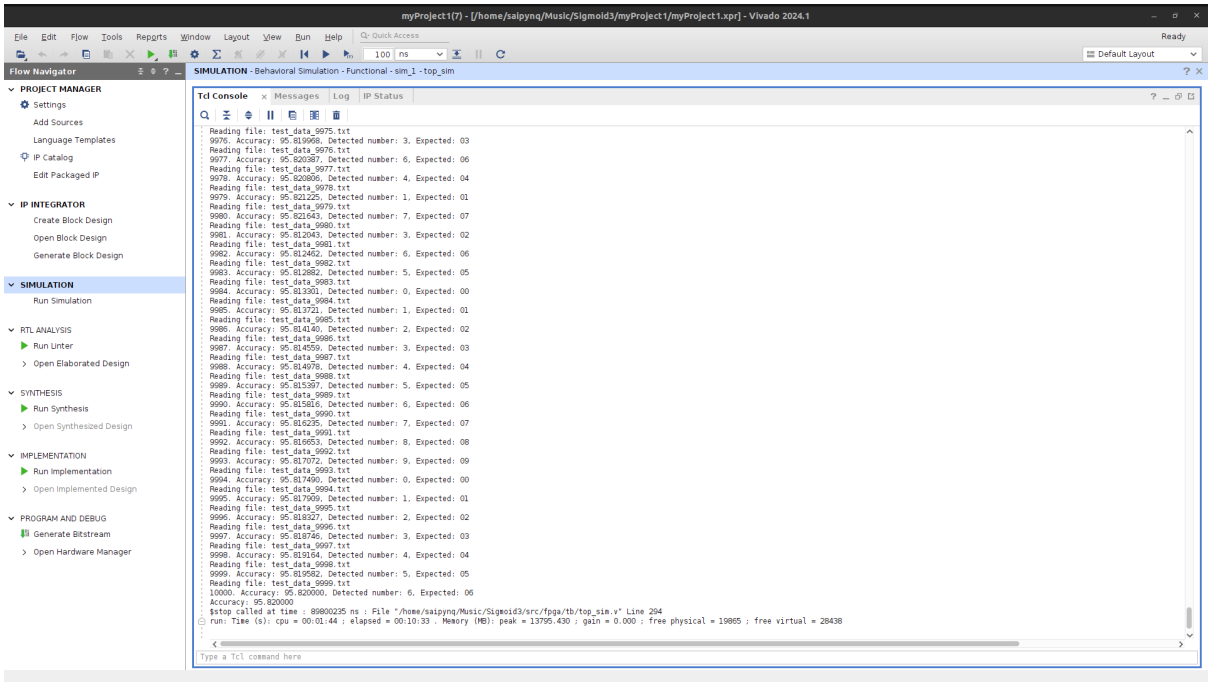

Figure 9.2: Simulation Output of MNIST MLP on FPGA

Explanation of Simulation Output:

The simulation output shown above demonstrates the accuracy and predictions for each test sample from the MNIST dataset. Key observations include:

- The console logs display each test sample's predicted number along with the expected (true) label. This format allows us to verify the accuracy of the MLP in simulation before deploying it to hardware.
- Accuracy values are displayed for each prediction, providing insight into the model's performance.
- The total runtime, memory usage, and peak resource usage are also reported, which are helpful for analyzing the design's efficiency and feasibility for real-time deployment.

This simulation output indicates that the design can achieve high accuracy on the MNIST dataset, meeting our initial expectations for inference on FPGA. Future steps include implementing the bitstream on the PYNQ-Z2 hardware and performing real-time testing.

—

This section provides a comprehensive overview of the MNIST MLP implementation on FPGA using ZyNet, along with detailed explanations of the block design and current simulation results.

Chapter 10

Conclusion

Conclusion and Future Work

In this M.Tech Phase 1 project, we explored the setup, configuration, and practical application of FPGA-based systems using the PYNQ-Z2 board. The project began with an introduction to Field-Programmable Gate Arrays (FPGAs), their architecture, and the benefits they offer over traditional CPU-based systems. We examined the integration of programmable logic (PL) and processing systems (PS) on the Zynq 7000 SoC, facilitating hardware-software co-design that leverages both ARM Cortex-A9 processors and FPGA fabric.

Through various implementations such as simple adders, FIR filters, and image dimming applications, we validated the capabilities of FPGAs for performing tasks with greater efficiency than general-purpose CPUs. Additionally, we reviewed the simulation results of running a neural network on the FPGA, which illustrated the potential for high-speed, parallel processing and low-latency computation.

The primary objective of MTP-1 was to build a comprehensive understanding of FPGA-based development, focusing on the data flow between the PS and PL, and the effective interfacing required for high-performance applications. The completed work lays a strong foundation for more complex implementations and showcases the suitability of FPGAs for accelerating specific workloads.

Future Work

The next phase (MTP-2) will delve into more resource-intensive applications and extend the scope of FPGA utilization for real-time machine learning tasks. The overarching goal is to develop an embedded Linux-based platform functioning as an AI-accelerated Network Interface Card (NIC). This platform will conduct real-time network traffic analytics, enabling efficient monitoring, anomaly detection, and traffic type prediction.

Future work will focus on:

– Implementing advanced deep learning models tailored for network traffic analysis.

- Optimizing FPGA resource allocation for complex AI workloads, ensuring minimal latency and maximal throughput.
- Integrating scalable solutions adaptable to evolving network demands, enhancing the platform's robustness.
- Demonstrating the performance improvements and responsiveness gains compared to traditional CPU and GPU approaches.

This forward-looking approach aims to validate the potential of FPGA-accelerated platforms for enhancing performance in machine learning-driven analytics, offering an adaptable, high-performance solution for modern network challenges.