TECHNISCHE UNIVERSITÄT
CHEMNITZ

Evaluation of AXI-Interfaces for Hardware Software Communication

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc. in Embedded Systems

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Ankit Sharma
Student ID: 425911
Date: 19.12.2018

Supervising tutor: Prof. Dr. W. Hardt
Dipl. Inf. René Schmidt

Abstract

A SoC design approach is implemented for the MERGE project which features Machine Learning (ML) interface for the hardware design. This setup deals with detection and localization of impact on a piezo metal composite. Development of the project is executed on Digilent ZYBO board. ZYBO incorporates Xilinx ZYNQ architecture. This architecture provides Processing System (PS) and Programmable Logic (PL) that communicate with each other via AMBA Standard AXI4 Interface.

Communication cost have major influence on the system performance. A optimized hardware software partitioning solution will reduce the communication costs. Therefore, best fitting interface for the provided design is needed to be evaluated to trade-off between cost and performance. High performance of AXI Interface will provide efficient localization of impact, especially for real-time scenario. In the thesis, the performance of three different AXI4 interface are evaluated. Evaluation is performed on the basis of the amount of data transferred and the time taken to process it.

Evaluation of interfaces are done through implementation of test cases in Xilinx SDK. Hardware design for AXI4-Interfaces is implemented in Vivado and later tested on Digilent ZYBO board. To test the performance of interfaces, read and write operations are initiated by PS on interface design. Each operation is performed for multiple data lengths. Average execution time is calculated that highlights time taken to transfer the corresponding input data length.

Through these tests, it is found that AXI4-Stream is the best choice for a continuous set of data. Preferably, it provides unlimited burst length which is useful for the current project. Among other two interfaces, AXI4-Full performed better in terms of execution time as compared to AXI4-Lite.

Keywords: AXI4-Full, AXI4-Lite, AXI4-Stream, Vivado, ZYNQ

Contents

List of Figures

List of Tables

List of Abbreviations

AHB Advanced High-Performance Bus

AMBA Advanced Microcontroller Bus Architecture

APU Application Processor Unit

APB Advanced Peripheral Bus

ARM Advanced RISC Machine

ASB Advanced System Bus

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

CAN Controller Area Network

DMA Dynamic Memory Access

FPU Floating Point Unit

I2C Inter-Integrated Circuit

IP Intellectual Property

MMU Memory Management Unit

NoC Network On Chip

OCM On-Chip Memory

OP-AMP Operational Amplifier

PLL Phase Locked Loop

PL Programmable Logic

PS Processing System

ROM Read Only Memory

SDK Software Development Kit

SDIO Secure Digital Input/Output

SoC System on Chip

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

Acknowledgments

I would like to offer my gratitude towards Chair of Computer Engineering and people involved in the chair.

First and foremost, I am thankful to Prof. Dr. W. Hardt to provide me an opportunity to do master thesis in the Professorship of Computer Engineering.

I, sincerely, like to thank my supervisor, Dipl. Inf. René Schmidt, for supporting me throughout the journey. I am grateful to have him as my supervisor since I always felt motivated and happy while working under him.

1. Introduction

1.1. Motivation

In context of MERGE technologies, "Technology Fusion for Lightweight Structures" aims to develop a novel touch-based user interface for a piezo metal composite. Overall concept of the project is illustrated in Figure 1.1.

Piezo foil is glued to a metal sheet with copper electrodes on it (step 3 and 4). These electrodes are polarized to amplify the resulting voltage from piezo film[37]. Afterwards, composite can be formed to give any required shape. This thesis is concerned with steps 7 and 8, highlighted in a rectangular box, that are responsible for detection and localization of impact on the composite.

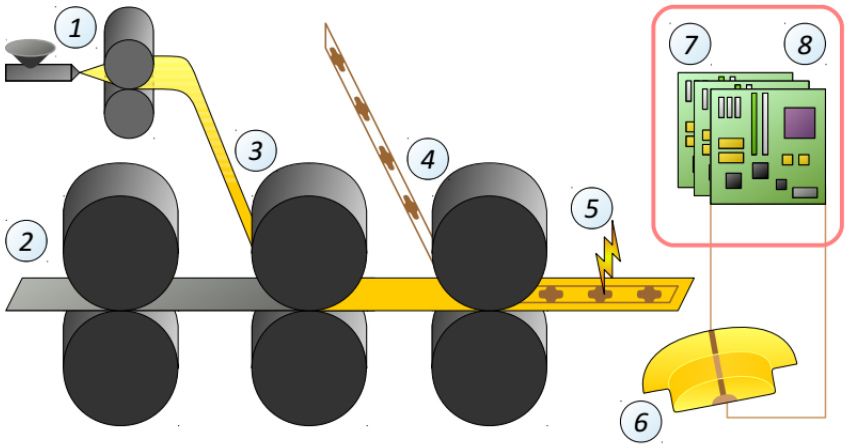


Figure 1.1.: Processing chain of Piezo Metal Composite[37]

It requires embedded signal processing to process piezo signals and extract appropriate features for classifier to accurately localize the point of impact. Since composite is allowed to take multiple form, it is viable to use a Machine Learning (ML) interface. Classifier's accuracy depends on the type and amount of training data used for learning rather than geometry of the composite. Hence, it will eliminate dependency of the results on mechanical parameters such as changes in propagation of mechanical waves due to changes in physical shape of the composite[37].

Signal processing has been implemented on FPGA. However, it is not feasible to run ML part on FPGA, therefore, it demands a hardware/software co-design approach to run machine learning algorithm on the processor.

Current design as depicted in Figure 1.2 uses Digilent ZYBO (ZYnq BOard) board because of its inclusion of ARM-based processor with an FPGA on a Zynq System-on-Chip (SoC). Moreover, for development purpose, for a low-cost it offers flexible design.

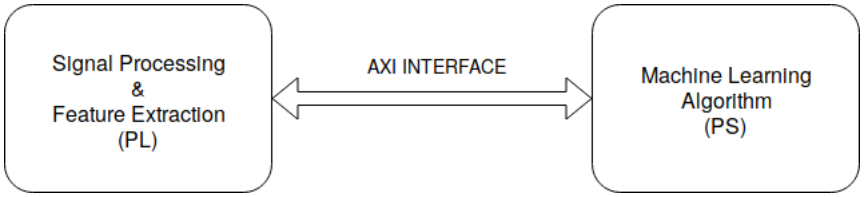


Figure 1.2.: Block Diagram of the current design

As seen from Figure 1.2, Zynq SoC comprise AXI Interface to link Programmable Logic (PL) and Processing System (PS). AXI Interface provides several configurations that can have impact on the overall speed of the design. In order to provide a optimized hardware software partitioning solution, the communication costs have a major influence on the system performance. Therefore, an analysis of the AXI-Interfaces representing the communication costs is mandatory.

1.2. Structure of the Thesis

Thesis is structured as follows:

Chapter 2, State of the Art, lists some of the communication protocols that are used in FPGA design. Since work in the thesis is done on SoC, therefore, various SoC communication standards are discussed. Among these, Zynq architecture is explained in detail because of its use in the implementation. Lastly, for the evaluation of AXI Interfaces, current hardware and software designs are mentioned.

Chapter 3, Concept, compares various SoC communication standards to provide an overview of features supported by such standards. In addition, approach taken to evaluate AXI-Interfaces and platform used for such purpose are discussed.

Chapter 4, Implementation, details FPGA implementation, provided in the thesis, for signal processing and feature extraction. It also explains implemented design, both hardware and software, for evaluation.

Chapter 5, Results and Discussions, highlights the performance of AXI Interfaces and examines their results to find the best fitting interface for given FPGA implementation and machine learning interface.

Chapter 6, Conclusion and Future Work, summarizes the thesis through its implemented design and evaluation results. Based on thesis's conclusion, possible ideas that can be implemented to improve the design are mentioned.

2. State of the Art

2.1. Possible Communication Alternatives for FPGA

There have been several communication protocols where are used in FPGA designs such as Universal Serial Bus (USB), I2C (Inter-Integrated Circuit), Serial Peripheral Interface (SPI), etc. In this chapter, some of the widely used communication protocols are discussed. These include: SPI, I2C, CAN, USB, and TIA/EIA standards. These protocols can be implemented within FPGA design as an IP Core. Some can be used to communicate with FPGA, for example, SPI.

These communication protocols uses serial communication. Through the discussion, it will be easier to have an outlook of the requirements for each protocols and complexity involved in using it. Moreover, it will provide an overview about features of various protocols.

Some of the well known standards which are used for the communication between FPGA and microprocessor are listed below:

2.1.1. Serial Peripheral Interface

Serial Peripheral Interface (SPI), developed by Motorola, is a synchronous serial bus that provides full duplex communication between the master and one or more slaves. It is well suited for communication between integrated circuits for low/medium data transfer speed with on-board peripherals. SPI bus consists of four wires as depicted in Figure 2.1. A separate Slave Select (SS) signal is needed for each slave which adds to extra wiring.

Functions of these four signals is as follows. Master drives the clock signal (SCLK) for the slaves. To send data from master to slave, MOSI (Master Output Slave Input) pin is used. For sending data from slave to master, slave uses MISO (Master Input Slave Output) pin. Master selects appropriate slave with the help of SS (Slave Select) pin and then sends/receives the data.

When used in full duplex mode, SPI Interface can achieve data rates of upto 1 Mbps. Therefore, it is well suited for low speed communications, for example, for configuring FPGAs where a microprocessor reads a bitstream file via SPI interface and sends it to FPGA over slave serial interface.

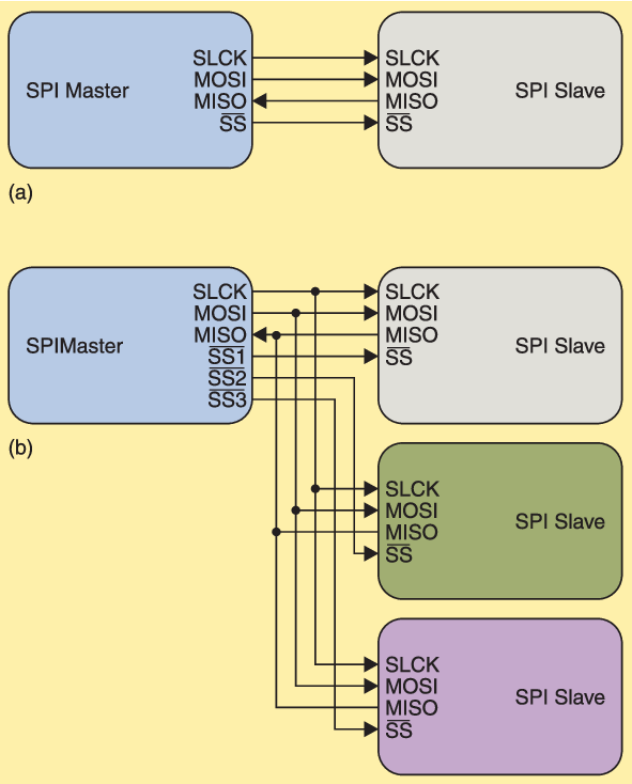


Figure 2.1.: SPI Bus Topologies[3]

2.1.2. Inter-Integrated Circuit

Phillips introduced Inter-Integrated Circuit (I2C) protocol in 1982 for serial communication between ICs placed on a same board. I2C protocol, similar to UART (Univeral Asynchronous Receiver Transmitter), supports two lines for communication, Serial Clock (SCL) and Serial Data (SDA), which handles multiple masters and slaves as depicted in Figure 2.3a.

In I2C bus configuration, SCL and SDL lines are connected to positive supply voltage using a pull-up resistor. Hence, a device on a I2C bus uses an open-drain approach, i.e., it can only drive a logic 0 (LOW) on a bus.

Master and slave communicate via 7-bit/10-bit address. Master sends address to slaves in a message frame coupled with control information as shown in Figure 2.3b. Communication starts and ends with special bits supported by I2C protocol. Each address is followed by a read/write information and an acknowledgment from the receiver.

I2C bus comes in various speeds as needed for the application. A unidirectionl bus supports Ultra Fast-mode with a bit rate up to 5 Mbps whereas a bidirectional bus supports:[29]

- Standard-mode (0 - 100 kbit/s)
- Fast-mode (0 - 400 kbit/s)

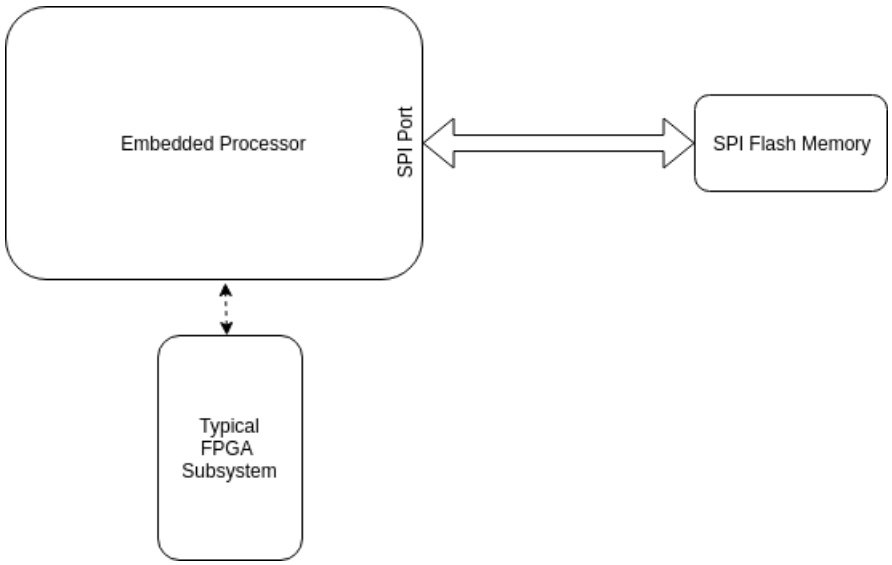
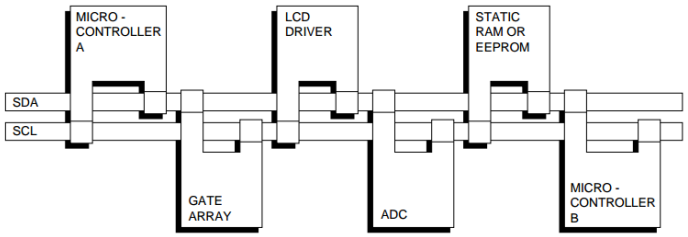


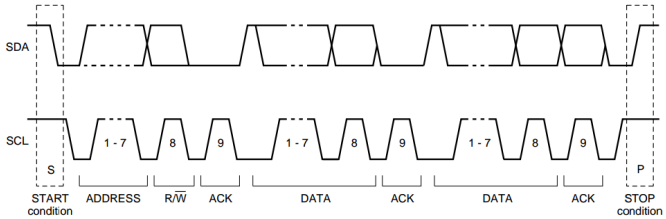
Figure 2.2.: Typical Embedded Microprocessor System with an FPGA[2]

- Fast-mode Plus (0 - 1 Mbit/s)
- High-speed mode (0 - 3.4 Mbit/s)

I2C protocol is mainly used for low speed communication and control applications such as in EEPROM (Electrically-Erasable Programmable Read-Only Memory), ADC (Analog-to-Digital Converter), microcontroller, LCD (Liquid Crystal Display) controllers [25]. Moreover, the bus is used in various control architectures such as System Management Bus (SMBus) and Power Management Bus (PMBus)[29].



(a) I2C bus configuration using two microcontrollers



(b) Communication frame in I2C data transfer

Figure 2.3.: I2C Bus Configuration and Communication Frame[29]

2.1.3. Controller Area Network

In 1986 Robert Bosch GmBH, at the Society of Automotive Engineers (SAE) conference, publicly released a serial communication bus, Controller Area Network (CAN)[10]. Reason behind the development of serial network protocol was to have an efficient real-time based communication between various electronic devices in automotive applications.

CAN architecture replaced complex point-to-point wiring harness between various Electronic Control Units (ECUs) by a two-wire bus. These two wires, CANH and CANL, are configured in a twisted-pair cable which cancels out electromagnetic interference. CAN bus provides two logic states: a recessive state and a dominant state as illustrated in Figure 2.4. A dominant in a bus denotes a differential voltage between CANH and CANL of 2V (logical HIGH) whereas a recessive signifies a zero differential voltage.

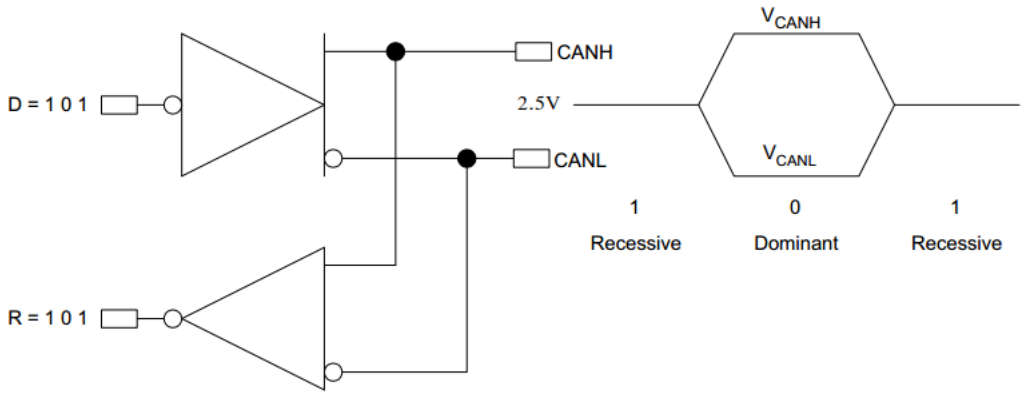


Figure 2.4.: Inverted logic of a CAN bus[12]

CAN devices (or nodes) access CAN bus through carrier-sense, multiple-access protocol with collision detection and arbitration on message priority[12]. Prior to transmission of data, availability of bus is checked by a node. If multiple nodes transmit data at the same time, then bit-wise arbitration scheme is employed to resolve the conflict. Arbitration is performed on the value of message identifier stored in the arbitration field (Figure 2.5). Priority of a message is inversely proportional to the value of an identifier. So, among multiple frames, a node frame of lowest identifier wins the arbitration and, therefore, is transmitted on the bus. As depicted in Figure 2.5, CAN specification provides two different identifiers, 11-bit (standard) and 29-bit (extended), leading to an increase in the maximum number of nodes allowed on the bus. Message transfer between CAN devices is controlled by four frames[8]: a data frame, a remote frame, an error frame, and an overload frame. A data frame contains 0-8 bytes of data on the bus. Data can be requested from a specific node through a remote frame which contains no data. An error frame is transmitted by a node in case of an error in the bus. Overload frames provides additional delay between data or remote frames.

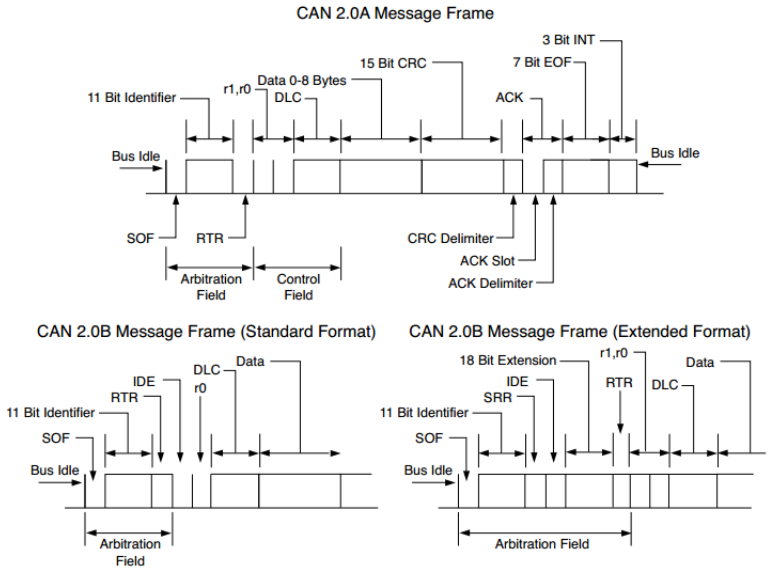


Figure 2.5.: CAN frames[12]

fig:canspeed it is seen that CAN bus allows a maximum speed of 1 M bits/s for a bus length of 40 m. Based on rate of transmission, CAN is divided into three types[11]

- High Speed CAN (Up to 1 M bits/s)
- Low Speed/Fault-tolerant CAN (Up to 125 K bits/s)
- Single Wire (Up to 83.3 K bits/s)

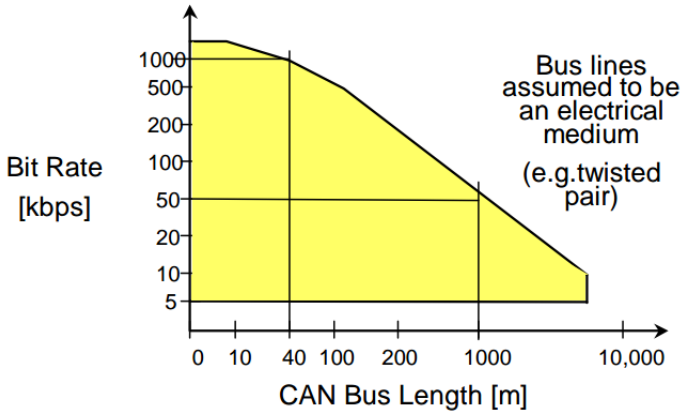


Figure 2.6.: Relation between Baud rate and CAN bus length[34]

Because of its low cost, robust noise immunity and real-time capabilities, CAN protocol is utilized in variety of applications apart from automotive. CAN finds its usage in textile machinery, medical devices, and as production line equipment[11].

2.1.4. Universal Serial Bus

In 1990s, growing use of personal computer (PC) led to development of various peripherals such as mouse, keyboard, modem, printer etc. Separate interfaces such as serial and parallel port were used for the attachment of such peripherals to PC. From user's point of view, there was a need to replace different connectors by a single interface that can allow multiple devices to communicate with each other. Therefore, in 1994, Universal Serial Bus (USB) standard was developed through a joint effort of multiple industries: Compaq, IBM, DEC, Microsoft, Nortal, and NEC[28].

USB architecture consists of a host computer connected to wide range of peripherals through a cable bus. Attached peripherals share the bus bandwidth among them and are allowed to attach or detach while the bus is in operation[9]. These peripherals are connected to USB host in a tiered-star topology as depicted in Figure 2.7. USB protocol supports a maximum of 127 devices that can be attached to a single host controller. USB is a polled bus, that is, only the host controller can initiate a transfer in the bus.

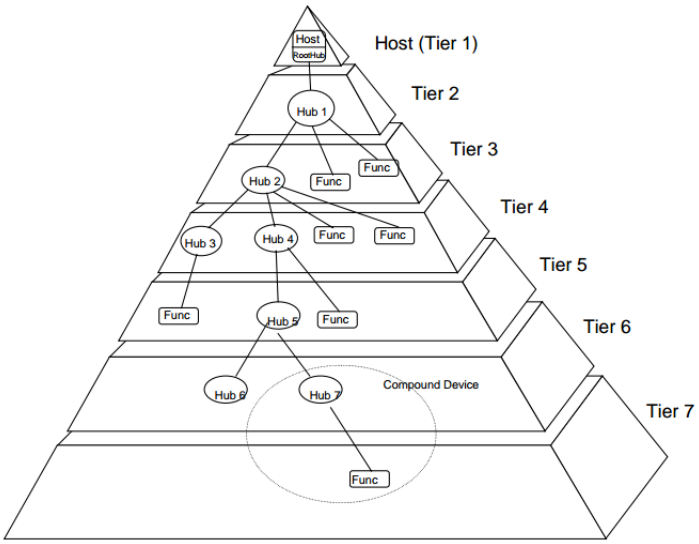


Figure 2.7.: USB bus tiered-star topology[e][9]

Communication between host controller and USB device is done via pipes (Figure 2.8). An Endpoint in a USB device is used as a storage for data, which can be addressable by the host controller. From Figure 2.7, data pipes, as the name suggests, are unidirectional and carry data. Each USB device have a bidirectional control pipe to configure device or provide status and control information.

Each transaction in the bus is initiated by the host controller through the token packet. This packet contains information about the type and direction of transaction, the USB device address, and endpoint number[9]. After the token packet, host can send or receive data packets. USB device transmits a handshake packet to denote a successful transfer.

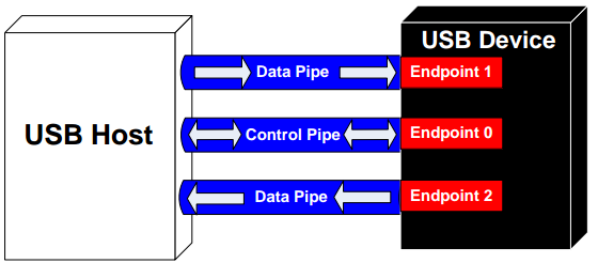


Figure 2.8.: USB Pipe Model[28]

USB Specification provides four data transfer types[9]

- Control Transfers: Used for configuration of the device.
- Bulk Data Transfers: Used to transfer of large amount of data without guarantee on transfer speed or latency.
- Interrupt Data Transfers: Used to transfer small amount of data though in timely manner.
- Isochronous Data Transfers: Used for guaranteed data delivery rate through assurance of set amount of bus bandwidth and latency.

To carry out data transfers, USB cable uses differential signaling (Figure 2.9. The clock is encoded with the differential data and uses Non-Return-To-Zero Inverted (NRZI) scheme[9]. Bit stuffing is employed to maintain synchronization between sender and receiver, in cases where there are no transitions in the data.

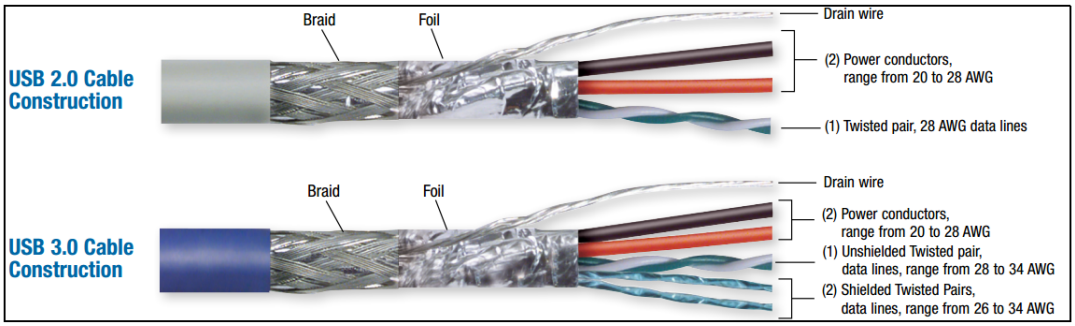


Figure 2.9.: USB Cable[26]

From Figure 2.9 we see that there a two variants of cables used for USB bus communication. USB 2.0 supports three speeds: High-speed (480 Mbits/s), Full-speed (12 Mbits/s), and Low-speed (1.5 Mbits/s). USB 3.0 is back compatible with USB 2.0 devices and, in addition, provides a Superspeed bus that supports transfer rate of up to 5.0 Gbits/s[2]. A successor to 3.0, USB 3.1 scaled the Superspeed to 10.0 Gbits/s[2].

Availability of such transfer speeds allow USB to be employed in different applications than just being a way to connect peripherals to PCs. It enables faster performance in downloading data or video[26]. USB, as a protocol, is used in non-traditional application such as industrial automation[2]. Moreover, USB is used a power source for charging mobile devices.

2.1.5. TIA/EIA Standards

Telecommunication Industry Association/Electronic Industries Association (TIA/EIA) defined physical layer standards intended for serial communication. These standards provide electrical and mechanical specification that can be utilized by a serial protocol.

For example, UART (Universal Asynchronous Receiver Transmitter) hardware is used in conjunction with these standards to facilitate serial communication. Another example is Modbus serial protocol used with RS-232 or RS-485 interfaces.

These standards are also referred with a prefix "RS" (Recommended Standard) such as RS-232, RS-422, and RS-485.

RS-232 (TIA/EIA-232)

RS-232 is a single-ended, full-duplex, communication interface between a driver and a receiver in the point-to-point configuration. This interface was introduced by EIA in 1962 to standardize communication between Data Transmission Equipment (DTE) such as a PC and Data Communication Equipment (DCE) such as a modem[20].

For communication purpose, RS-232 standard also defined a standard connector, named DB-25 (Figure 2.11). However, when IBM PC AT was released in 1984, it featured a 9-pin D type connector which later became common in use as an interface between data acquisition devices and computer systems.

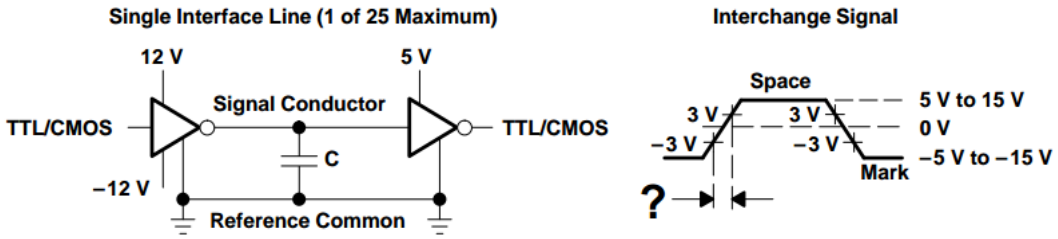


Figure 2.10.: TIA/EIA-232-F Electrical Specification[20]

Voltage levels used by RS-232 are specified in Figure 2.10. Voltage levels between ±3V are undefined such that effect of noise at receiver's end is reduced. However, use of single-ended configuration is prone to common-mode noise. A shift in ground level at receiver's end can lead to unbalanced situation. Use of a common ground signal limits the maximum data throughput.

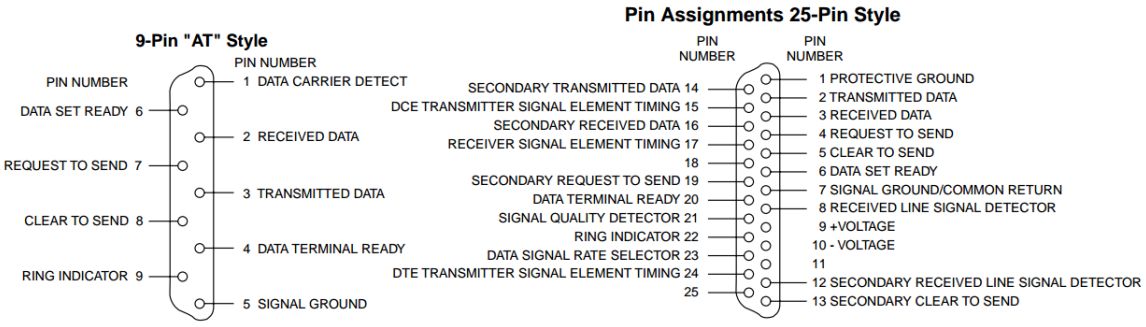


Figure 2.11.: RS-232 connectors[31]

RS-232 standard defines a maximum data rate of 20 kbits/s for a maximum cable length of about 15 to 20 meters[20]. It states that the cable is allowed to have a maximum capacitance of 2500 pF.

RS-422 (TIA/EIA-422)

RS-422 was introduced to overcome noise issue of RS-232 because of single-ended interface[23]. RS-422 standard provides electrical specification for employment of balanced data transmission over long distances.

It is a simplex multidrop standard, that is, on a shared bus only single driver and multiple receivers (up to 10) can exist. Similar to RS-232, it can also be used in point-to-point configuration.

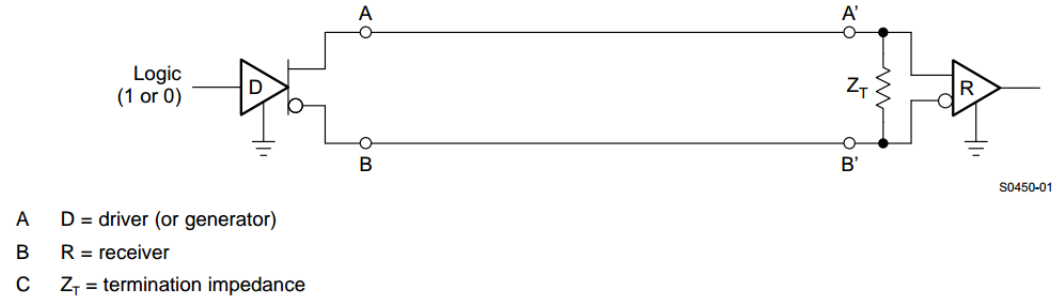


Figure 2.12.: RS-422 Balanced Voltage Circuit[22]

An overview of the interface circuit is depicted in Figure 2.12. As per the specification, termination impedance is required to reduce reflection caused by mismatch of characteristic impedance of the line.

RS-422 allows a maximum cable length of 1200 meter, however, it would not result in maximum data rate as depicted in Figure 2.13. Voltage range for RS-422-compliant driver lies in the range of $\pm 6V$ whereas for receiver, it is $\pm$10V with a threshold level of 200 mV[22].

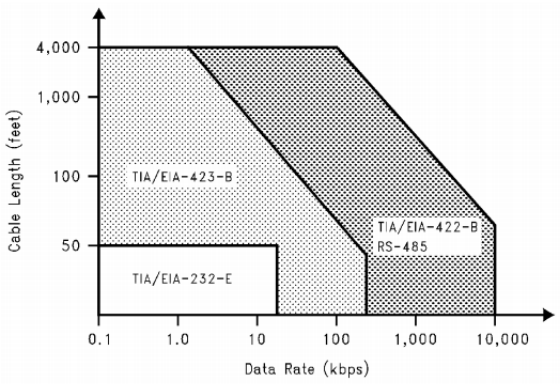


Figure 2.13.: Comparison of cable length and data rate[23]

RS-485 (TIA/EIA-485)

In comparison to RS-232 and RS-422, RS-485 is most widely used standard in industry owing to its balanced digital transmission line with an interconnection of multiple transmitters and receivers. RS-485 uses a half-duplex communication scheme in multi-point configuration as seen inFigure 2.14. It can transmit data over distances of several kilometers with fewer noise emission.

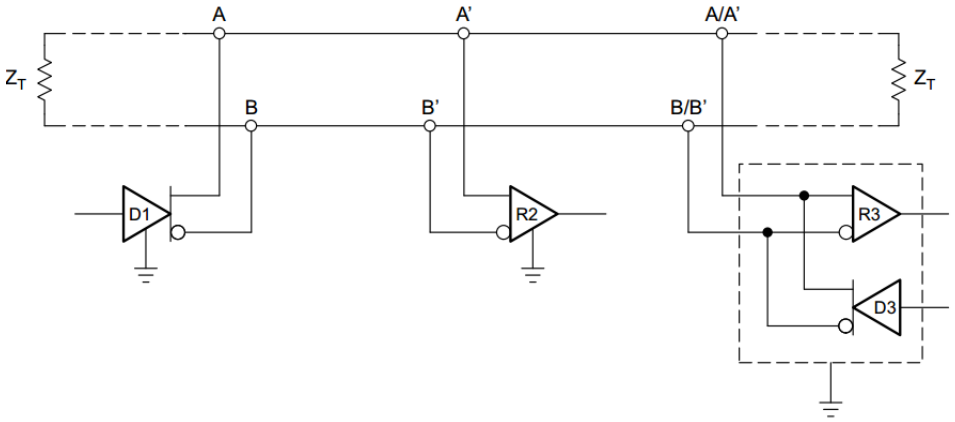


Figure 2.14.: RS-485 Balance Voltage Circuit[22]

RS-485 specification defines only electrical layer and its electrical specification is compliant with RS-422 drivers and receivers[22]. RS-485 compliant drivers and receivers voltage on the lines range from -7 V to 12 V. In contrast to RS-422, it requires termination impedance at both ends of the cable.

As per the specification, maximum cable length can be 1200 meter and a high signaling rate of up to 50 Mbits/s is achievable[21]. Variation of data rate with respect to length of the cable is depicted in Figure 2.13.

Due to its low noise coupling and various other advantages, various standards committees incorporated RS-485 as the physical layer specification for their com-

munication standard such as the Profibus standard and Small Computer Systems Interface (SCSI)[21].

2.2. Outline of SoC Bus Standards

Various computer components such as processor cores, on-chip memory, external memory interface, Digital Signal Processor (DSP), input/output devices, etc., are placed on a single chip called System-on-Chip (SoC). Compared to motherboards such arrangement leads to reduced board space and less power consumption. Owing to this they mainly find their use in mobile computing devices.

In the thesis, we will look into SoC that integrates FPGA (reconfigurable SoC) since implementation is performed on such system. In contract to Application-Specific Integrated Circuit (ASIC), rSoC provides more flexibility with short turn around time for testing the hardware design and is generally cost-effective. However, ASICs are aimed to comply with a particular purpose, hence, they acquire less space than FPGAs and have lesser power requirements.

IP (Intellectual Property) cores for SoC are designed with different interfaces and communication protocols. SoC needs minimal glue logic to integrate them in a chip and so it resulted in development of on-chip bus standards[27]. There have been several interconnect topology to interface various IPs in SoC such as point-to-point, bus system, crossbar switch, Network-on-Chip (NoC), etc. Among these, Network-on-Chip provides better design scalability and uses wires efficiently[16]. It makes use of data packets for communication between devices similar to TCP/IP (Transport Control Protocol/Internet Protocol) model.

To increase the overall speed of SoC designs, some emerging on-chip interconnect technologies such as optical interconnects, Radio Frequency (RF) based interconnect, and CNT (Carbon Nanotube) interconnect, are in experimental phase[33].

In this section, we will look into various on-chip bus standards which are flexible and robust enough to integrate numerous functional units on a single chip. Bus architecture is widely accepted integration technique for inter-module communication in SoC. It is based on traditional address and data buses used in computers to carry information across the system.

2.2.1. AMBA

A SoC interconnect standard named Advanced Microcontroller Bus Architecture (AMBA) was introduced by ARM in 1996 to facilitate communication between various IPs (Intellectual Property)[6]. It employs two layer hierarchical bus topology to interface SoC components based on their performance requirements.

High performance bus connects processor cores, memories, and other high-bandwidth components. Low performance bus provides interface to low bandwidth peripherals such as Ethernet, USB, UART, etc. It is a simple addressing bus with

latched address and control signals for low-speed peripherals. These buses are adjoined by a bridge as illustrated in Figure 2.15.

ARM AMBA system consists of: masters, slaves, bus arbiter, and central decoder. Multiple masters can connect to high-bandwidth bus whereas slave peripherals communicate with the system using low-bandwidth bus. Bus arbiter implements arbitration scheme to allow only one master to initiate transfer at the same time. Decoder provides select signals for peripherals based on address provided by the master.

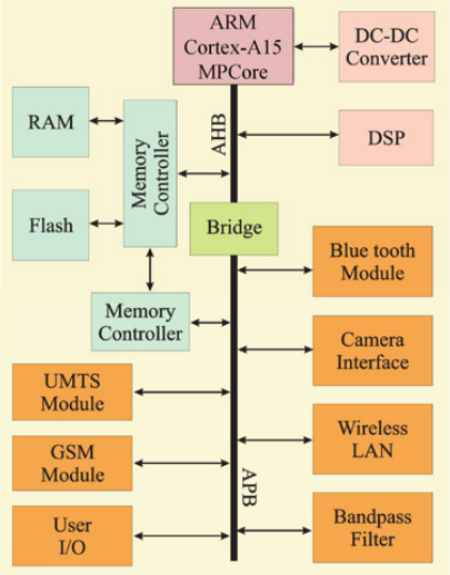


Figure 2.15.: AMBA-based mobile phone SoC[35]

The communications between IPs in an AMBA SoC is done in the form of transactions. A transaction, initiated by master, consists of securing the bus, commencing and completing read/write operation[35].

AMBA Specification version 1.0 released two buses: Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). Advanced System Bus is used as the main system bus for high-bandwidth components such as processor, memory, DMA controller, and APB Bridge. APB Bridge acts as a cross-over between ASB and APB and handles the bus handshake[6]. It supports multiple data bus widths: 8, 16, and 32 bits. APB bus provides a simple addressing with latched address and control signals. It is suitable for connecting peripherals such as UART, USB, Ethernet, General purpose input/output ports. It supports data width of 8, 16, 32 bits.

Advanced High-Performance Bus (AHB) was included in second version of AMBA. Similar to ASB, it provides pipelined operation but also supports burst transfer and split transactions. Multiple data bus widths allowed in AHB are 32/64/128 bits. It incorporates access protection scheme to distinguish between privileged and non-privileged access modes.

In third version (AMBA 3), AMBA added Advanced eXtensible Interface (AXI3)

which was later updated in fourth version to AXI4. AXI provides a point-to-point interconnect between master and slave which is advantageous over bus sharing. It enables higher bandwidth and lower latency in design. AMBA 3 also introduced Advanced Trace Bus (ATB) and AHB-Lite. ATB is part of CoreSight on-chip debug which is used for verification of an IP. AHB-Lite is a simplified version of AHB that allows only one master.

In addition to AXI3, AXI4 introduced two additional interfaces: AXI4-Lite and AXI4-Stream. AXI4-Lite is used for low-throughput memory mapped interface and provides powerful interface than APB. AXI4-Stream supports high-speed streaming of data[42]. AXI4 is discussed in greater detail in Section 2.3.

AMBA Specification version 4 (AMBA 4) released Advanced Coherency Extension (ACE) which is an extension to AXI4. It provides hardware management of cache memory across multiple cores. This removes the need of software to maintain coherency between caches, thereby, saving processor cycles and bus usage. A simplified version of ACE, ACE-Lite is introduced to provide one-way coherency for components that does not have their own cache memory like DMA (Direct Memory Access). Components with cache memory can also use ACE-Lite protocol when such cache is to be managed by an appropriate software.

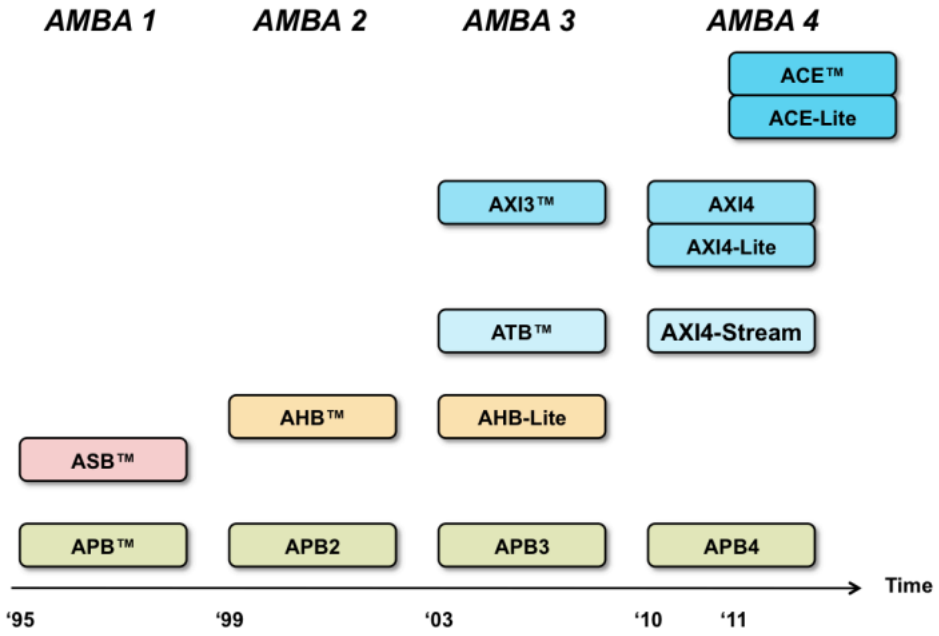


Figure 2.16.: AMBA Standards[36]

In 2013, AMBA 5 released Coherent Hub Interface (CHI) which is a complete redesign of ACE to incorporate increasing number of coherent clusters. Instead of using ACE Interconnect, it introduced a packed based layered architecture.

Since its inception, AMBA has been the de facto standard for creating SoCs. It is used by industries such as NVidia, Qualcomm, Actel, etc[35].

2.2.2. CoreConnect

IBM released on-chip bus standard named CoreConnect for interconnecting cores and enabling reuse of logical units in SoC through hierarchical bus system. It is composed of three buses for interconnecting cores, peripherals, and custom logic:

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)
- Data Control Register Bus (DCR)

Figure 2.17 illustrates the use of these buses in the architecture. PLB provides high-bandwidth and low latency with support for 16-, 32-, and 64-bits data transfers, extensible to 256-bits. PLB is intended to connect components that needs high throughput such as processor cores, DMA controllers, memories, etc. It is fully synchronous central arbitrated bus that supports up to 16 masters and unrestricted number of slaves. Both master and slave that are attached to PLB have separate bus for address, read data and write data. Hence, concurrent read and write transfers maximizes the bus utilization. Moreover, it features burst transfers, split transaction, and address pipelining which enhances bus throughput.

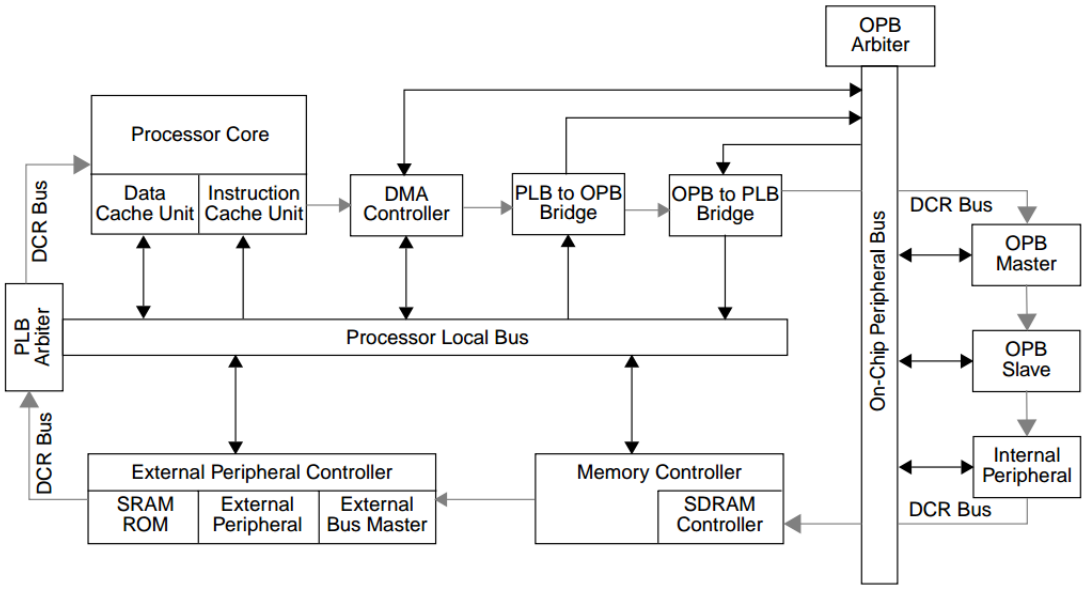


Figure 2.17.: CoreConnect Bus Architecture[19]

OPB reduces capacitive loading on PLB by providing independent support to low speed peripherals such as serial/parallel ports, USB, General purpose input/output ports, etc[c]. It uses separate 32-bit data and address bus. OPB features fully synchronous operation, burst transfers, single-cycle data transfers, multiple masters, and dynamic bus sizing. OPS supports multiple master devices through implementation of distributed multiplexer for both the address and data bus[19].

From Figure 2.17 it can be seen that communication between PLB and OPB is done through bridges. PLB masters use PLB to OPB bridge to write data to slave peripherals or use OPB to PLB bridge to read data from master peripherals.

DCR is a low performance, fully synchronous bus to provide status and configuration data across the system. It consists of 10-bit address bus and 32-bit data bus. Read or write transfer on the bus need minimum two cycles. Similar to OPB, it off-loads PLB from lower performance transfers by providing a daisy-chained connection between processor core master and other SoC devices acting as slaves. It makes use of ring topology which is implemented as distributed multiplexer across the chip. Such connection of SoC devices minimizes silicon usage[1].

Bus transaction is started by PLB which consists of address and data cycles as shown in Figure 2.18. Master requests for the bus by driving address and transfer qualifier signals. When PLB Arbiter grants bus to the master, these signals are passed to the requested slave in the transfer phase. Address cycle terminates when slave latches these signals in the address acknowledge phase.

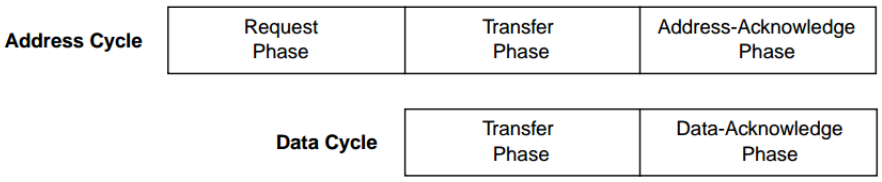


Figure 2.18.: PLB Address and Data Cycle[18]

Similarly master writes or reads data to/from slave in data cycle. Each data beat in data cycle consists: transfer phase where master drives write data or read data bus and data acknowledge phase which denotes end of the data cycle for the final beat of line transfer[1]. Maximum throughput occurs when data transfer and data acknowledge phases are concurrent.

CoreConnect finds its use in PowerPC400 family, for example, PowerPC 440 incorporates CoreConnect bus standard to interface components inside a PowerPC.

2.2.3. Wishbone

Silicore Corporation released an on-chip interconnect architecture named Wishbone bus which is supported by OpenCores, an organization that provides open source IP cores. Main objective for development of Wishbone bus was to have a common interface between IP Cores so as to enhance portability and reliability of the system[32].

Wishbone is a general purpose interface which provides set of signals and bus cycles for a single level bus. To connect IP Cores to bus, it defines two types of interfaces, namely, Master and Slave. IP cores that acts as master can initiate bus cycles whereas the ones which receive bus cycles are classified as Slave interfaces.

Major feature of Wishbone is the inclusion of possible interconnection architecture between Master and Slave interfaces as illustrated in Figure 2.19. These include: Point-to-Point, Data flow, Shared bus, and Crossbar switch.

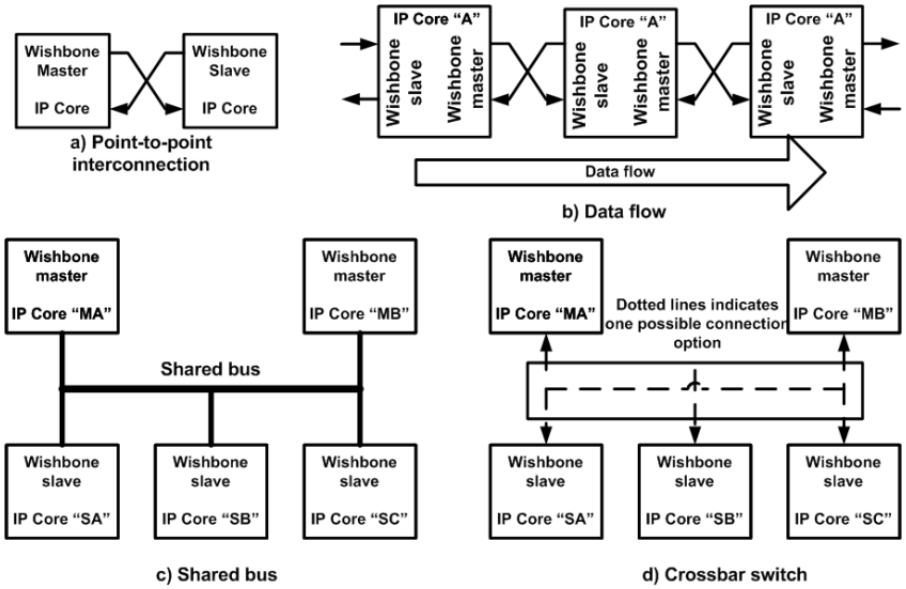


Figure 2.19.: Various Wishbone Interconnection Scheme[27]

Point-to-Point is the simplest of all interconnection scheme and falls short for SoC design where multiple masters have to be interfaced with multiple slaves.

Data flow architecture exhibits parallelism where each IP Core has both Master and Slave interface to provide the movement of data from core to core. This results in pipelining similar to the use Fetch-Execute-Write functional blocks in computer systems. Dataflow topology can find its in DSP algorithms where linear systolic array architecture is implemented[32].

Shared bus architecture interconnects multiple master and slaves through a shared bus. It needs bus arbiter and decoder logic to allow only one master to use the bus at a time. Compared to other architectures, it uses fewer logic gates and routing resources to interconnect SoC components which makes it relatively compact[32].

Crossbar switch is valuable when multi core SoC is used since it can speed up data transfer rate by utilizing multiple cores concurrently. Thereby, compared to shared bus architecture, it is faster but incorporates more interconnection logic and routing resources[32]. Moreover, use of a cross-bar interconnect system results in limited scalibility because of the use of centralized arbiter to control the bus[32].

Each Wishbone interface have set of signals which supports three types of bus cycles[32]

- Single Read/Write: Only single data transfer at a time.
- Block Read/Write: Multiple single read/write cycles.

- Read-Modify-Write: Allows multiple masters to share common slave by using semaphore bit.

Wishbone supports multiple data widths: 8, 16, 32, or 64 bits and address bus width from 1 to 64 bits.

2.2.4. Avalon

Altera Corporation, acquired by Intel, released Avalon bus architecture for System-on-a-programmable-chip (SPOC). Avalon makes use of multiplexers to drive the signals to appropriate peripheral. Hence, output of peripherals switch between HIGH and LOW without existing in a high impedance state. Signals and timings are defined in the architecture to allow master and slave components to communicate with each other[5].

From Figure 2.20 we see that Avalon features data-path multiplexing and multiple master. Masters and slaves interact with each other without the intervention of bus. However, arbitration is needed when multiple masters access the same slave. Arbitration technique used in Avalon bus is called slave-side (distributed) arbitration. This proves advantageous for this architecture since it enables multiple masters to perform transactions concurrently when same slave is not accessed during the same bus cycle[[32]].

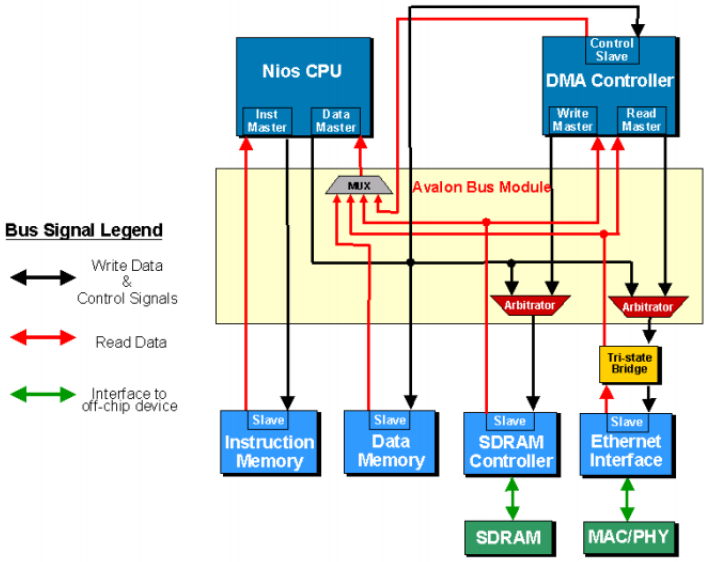


Figure 2.20.: Avalon bus based system[5]

Avalon bus stores logic and routing resources inside a Programmable Logic Device (PLD). It provides separate dedicated lines for address, data, and control. Therefore, there is no need of address or data decoder. Dynamic bus sizing is automatically implemented by Avalon bus. Avalon bus provides support for data width from 8 bits

up to 1024 bits[[24]]. High-bandwidth peripherals can use data streaming capability of Avalon bus.

2.2.5. Open Core Protocol

Accellera introduced Open Core Protocol (OCP) which defines a high performance and configurable interface for connecting IP Cores[a]. It is bus-independent, therefore, IP design with OCP interface can be reused. Its configurable interface enables IP core to use subsets of configuration features provided by OCP, thereby, optimizing die area.

From Figure 2.21 we see that OCP provides a point-to-point interface between IP cores and bus wrappers. In OCP system, master presents commands to slave and slave responds by accepting data from master or providing data to it.

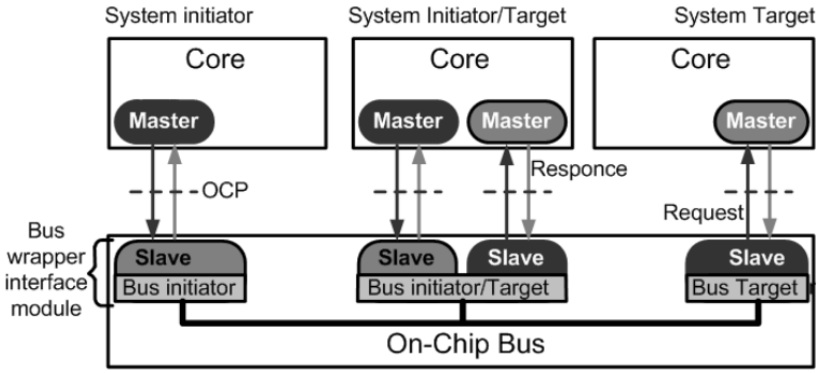


Figure 2.21.: System depicting Wrapped bus and OCP instances[27]

Some features of OCP interfaces are[4]: Fully synchronous, configurable address and data width, pipelined transfers, optional burst transfers, concurrent transfers via threads, allows master or slave to control transfer rate.

OCP includes signals such as data flow, control, verification, and test, needed for IP core's communication[4]. Moreover, apart from basic OCP used for interoperability, it provides multiple extensions[30] Simple Extension, Sideband Extension, and Complex Extension. Simple and Complex Extension support burst transfers. Complex Extension provides concurrency support for threads. Sideband Extension utilizes core specific user defined signals. Moreover, it supports JTAG (Joint Test Action Group), structured test environments, and clock control.

2.3. ZYNQ Architecture

In the thesis, evaluation of the design is performed on Z-7010 which is from the family of Xilinx ZYNQ-7000 SoC architecture. Next sections will give details about the architecture and list its features.

2.3.1. Overview

Xilinx ZYNQ-7000 SoC includes a Processing System (PS) and a Programmable Logic (PL) as displayed in Figure 2.23. PS consists of various functional blocks such as I/O peripherals, external memory interfaces, internal memory, and interconnect. Communication between PS and PL is performed through high-bandwidth AMBA AXI Interfaces.

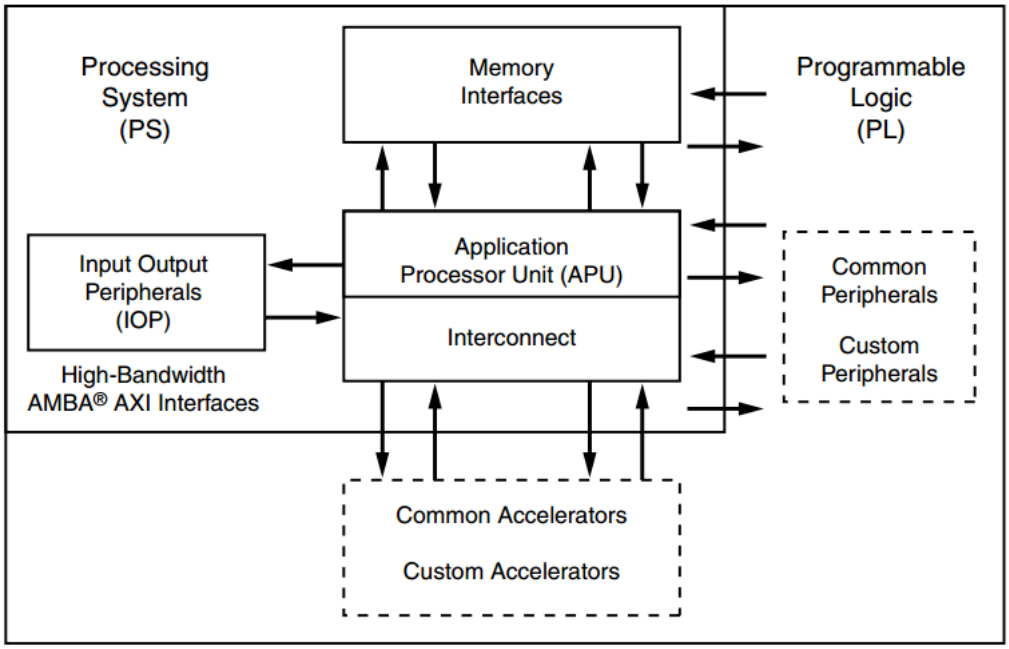


Figure 2.22.: High-Level Block Diagram[51]

This architecture scheme enables ZYNQ to utilize re-configurable functionality with PS ASIC design. PL can be used to run data-intensive task that supports parallelism whereas PS can be allowed to handle sequential process. This results in overall high throughput.

High bandwidth connectivity, with the help of AMBA Standard AXI4 Interfaces, is provided between PS and PL and also within PS to allow functional units to communicate efficiently.

Hardware design is loaded into PL fabric only after PS is configured. As the system starts, a hard coded boot ROM is executed (Boot Stage-0) which then allows PS to load First Stage Boot Loader (FSBL) from On-Chip Memory (OCM). Afterwards, second-stage bootloader can be used to load the kernel to DDR memory. Later, if needed, a bitstream file can be used to program the PL fabric.

Zynq-7000 family PL uses Xilinx 7 Series FPGAs such as Zynq device Z-7010, used in the thesis, contains Artix-7 FPGA. The clock to PL can originate from external clock pin or from PS. PS uses three PLLs (Phase-Locked Loop) and provides four input clocks to PL. Synchronization of clocks between PS and PL is managed by the architecture.

Zynq-7000 SoC devices are used in variety of fields such as Industrial motor control, Automotive and Medical diagnostics[52].

2.3.2. Features

Functional units integrated into ZYNQ-7000 are displayed in Figure 2.23. It compriseis of Processing System (PS) and Programmable Logic (PL).

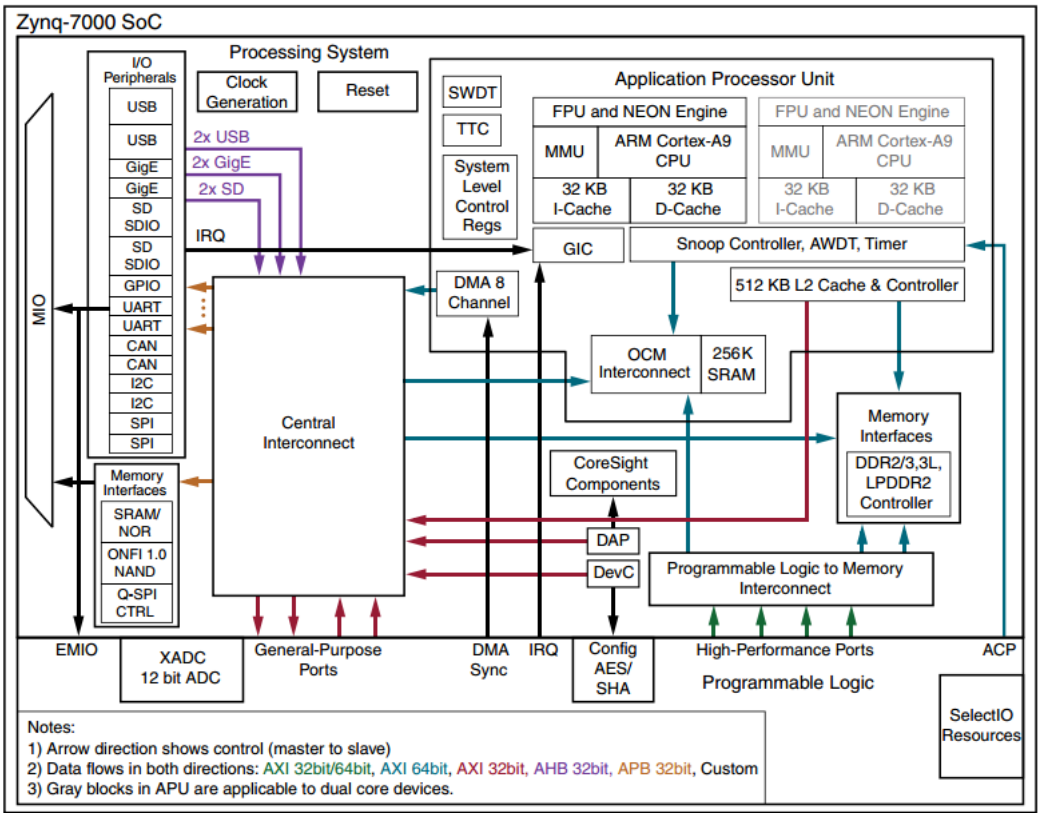


Figure 2.23.: Top View of ZYNQ Architecture[52]

Processing System consists of Application Processor Unit (APU) that embeds dual core ARM-Cortex A9 processor which comes with 32 KB Level-1 cache memory. It also provides Floating Point Unit (FPU) to facilitate floating operations needed in applications such as DSP. NEON Engine is featured to support Single Instruction Multiple Data (SIMD) instructions to improve the system performance through data parallelism. Other componenets in APU are DMA Controller and Memory Management Unit (MMU). Automatic cache coherency between processor cores is implicit in the processor design[43].

Central interconnect links APU to multiple peripherals. It is based on ARM NIC301 (Network Interface Configuration) design and uses 64-bit AXI Interface[52].

For debugging and tracing the software design, PS includes CoreSight controller. PS communicates to PL through two AXI based 32-bit General Purpose ports.

Different grade of ports are provided in the PL to communicate with the functional units in PS. Two 32-bit General Purpose ports are provided which are suitable for transfer of small amount of data such as control signals. There are two master and two slave GP ports. Four 64-bit High Performance ports which provides fast access to OCM and DDR memory. One Accelerator Coherency Port (ACP) is featured to access Snoop Control Unit (SCU) in APU. This enables PL to use OCM and Level-2 cache.

For processing Analog Mixed Signal(AMS), XADC (Xilinx Analog-to-Digital Converter) module is included in PL. It provides a dual 12-bit, 1 Mega sample per second ADC which is capable of accessing 17 external analog input channels[49].

Both PS and PL peripherals are mapped to System memory as shown in Figure 2.24. For unmapped peripherals in PS, Zynq features EMIO (Extended Multiplexed I/O) interface that allows these peripherals to use PL pins.

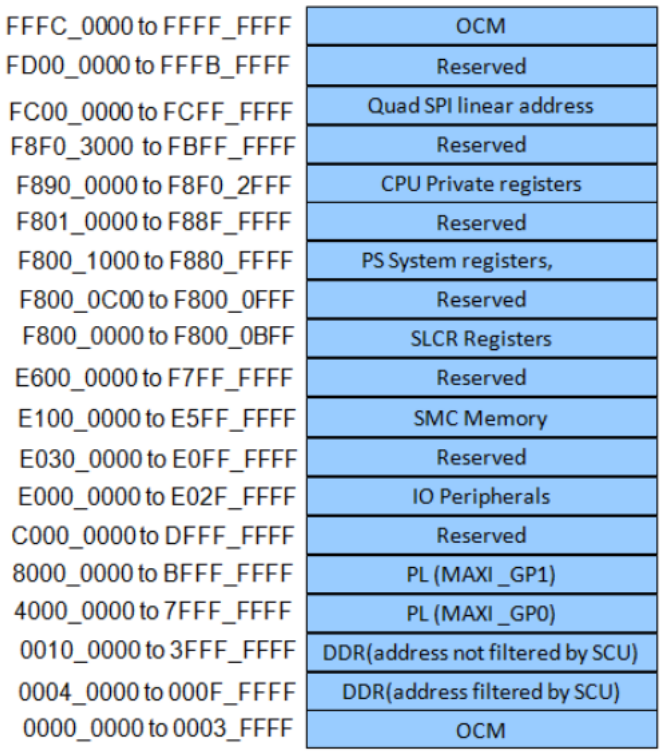


Figure 2.24.: System-Level Address map[43]

2.3.3. Communication Interfaces

Zynq-7000 family uses ARM AMBA4 AXI Interfaces for its architecture. AXI Interface implements point-to-point protocol between a AXI master interface and a AXI slave interface. AMBA4 lists three AXI interfaces as:

- AXI4 Memory Mapped Interfaces:

  - AXI4-Full Memory Mapped Interface
  - AXI4-Lite Memory Mapped Interface
- AXI4-Stream Interface

Each AXI interface defines a protocol that differs from each other in terms of performance and functionality. Depending on the application, a developer can select the suitable AXI protocol. This increases the flexibility of the design.

AXI4-Memory Mapped Interfaces

AXI4 Memory Mapped Interfaces use multiple channels to provide read/write access between the master and slave via AXI Interconnect. These channels are independent of each other and carry address, control, and data signals as illustrated in Figure 2.25.

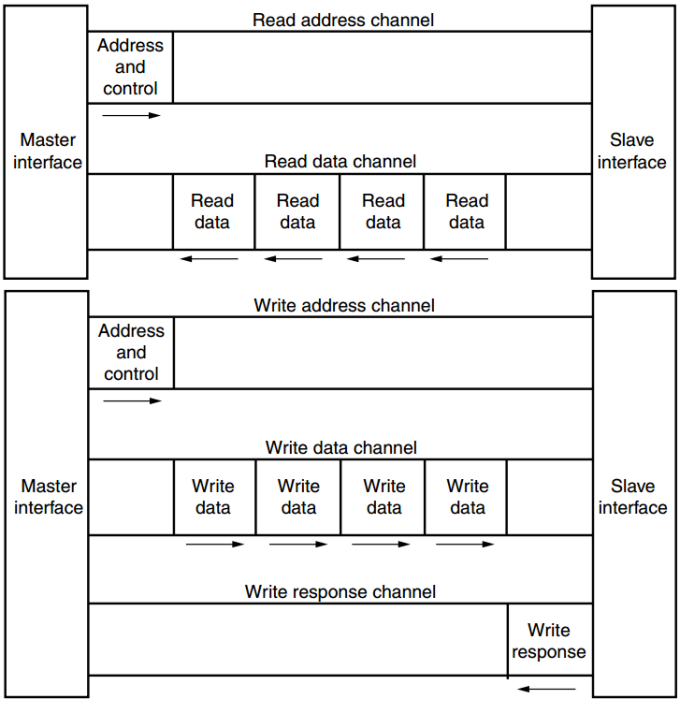


Figure 2.25.: Read and Write Channels[42]

Each channel makes use of two-way VALID/READY handshake mechanism before transmitting address/control or data signals. In Figure 2.26, information is transmitted only when both VALID and READY signals are HIGH.

Dependencies can exist between channel handshake signals which can lead to a deadlock condition. Therefore, AMBA AXI Protocol Specification[7] details dependency rules to be observed while implementing a design.

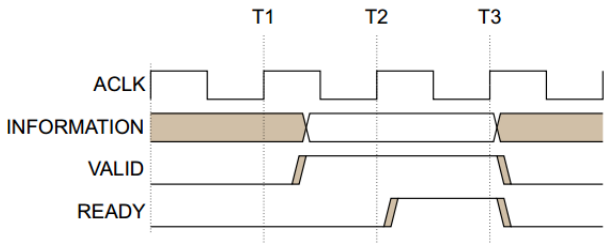


Figure 2.26.: Two-way VALID/READY Handshake[7]

Due to separate address and data channels for read/write, data transfer can be concurrent and bidirectional[42]. These channels are used in two types of memory mapped interfaces:

AXI4-Full Interface

AXI4-Full implements a burst-based, point-to-point protocol which provides various options to achieve high data throughput. AXI Master initiates a data transaction by putting first address of the burst transaction on the address channel. Based on burst parameters (size and length), slave is required to calculate the following transaction addresses. Burst size indicates the width of a data beat whereas burst length states the number of beats in a burst.

Some of the key features are mentioned as follows[7]. Burst length of 1 to 256 beats is supported for incremental bursts and 1 to 16 beats for wrap bursts. Variable data widths, ranging from 32-bits to 256-bits, is supported. Data can be upsized or downsized as required.

Multiple outstanding addresses can be issued. Transactions such as out-of-order and unaligned transfers are applicable which provides better overall throughput. Security features are also added for the interface such as read and write access protection. To increase performance, it has an option for addition of register slices in pipeline stages.

AXI4-Lite Interface

AXI4-Lite protocol is similar to AXI4-Full, however, it does not support burst transfers. In other words, it supports only a burst length of 1. It is mainly used to transmit control signals since they require only few clock cycles.

PS uses AXI-Lite protocol to configure an IP (Intellectual Property) through mapping it to the system address. For instance, if General Purpose (GP) AXI Port 0 is used to interface the IP, then address space for Port 0 will be used.

It supports a fixed data width : 32-bit or 64-bit. However, Xilinx IP supports only 32-bit wide data bus[43].

AXI4-Stream Interface

AXI4-Stream protocol defines a unidirectional flow of data from AXI master to AXI Slave without addresses (Figure 2.27. Instead of using addresses, it uses TID and TDEST signals to specify source and destination respectively[41]. It is mailny used for data-intensive applications where large amount of data has to be processed in the same manner such as in image processing, video streaming, etc.

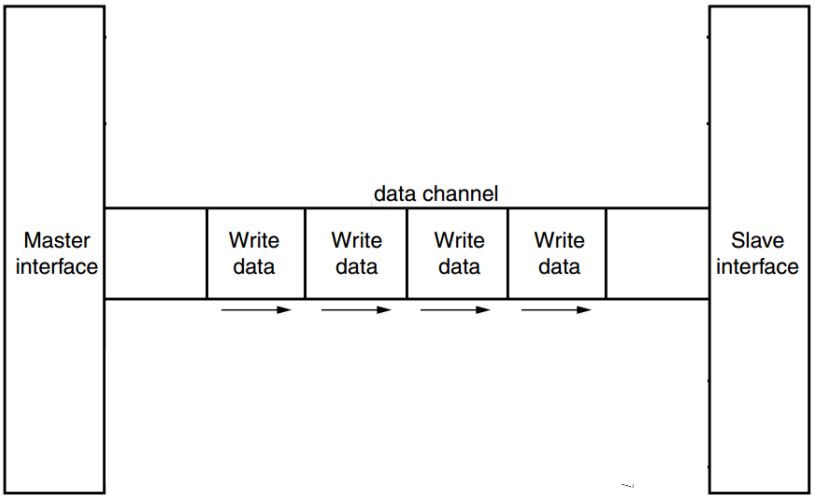


Figure 2.27.: AXI4-Stream Transfer via Single Channel[41]

AXI4-Stream protocols provides three byte types: Data byte, Position byte, and Null byte. Data byte contains data whereas Position byte acts as a placeholder in a data stream. Null byte neither contains data nor position information. According to [7], TKEEP signal is asserted to indicate a Position byte whereas for Null byte it is deasserted. TKEEP signal allows current byte to be kept/discarded from the stream.

Zynq-7000 provides four 32/64-bit AXI High-Performance (HP) ports which is mainly used to read/write OCM and DDR memory via AXI4-Stream protocol.

Some of the features provided by protocol[43]:

- Unlimited burst length.
- Sparse, continuous, aligned and unaligned streams.
- Transfer split, merge, interleave, upsize or downsize.
- Only ordered transfers allowed.

To layout a summary of AXI4 Interfaces, key differences in Table 2.1 are highlighted.

Interface	Features	IP Replacement
AXI4	◦ Memory mapped address/data interface ◦ Data burst support	PLBv3.4/v4.6, OPB, NPI, XCL
AXI4-Lite	◦ Memory based address/data interface ◦ Single data cycle only	PLBv4.6, DCR, DRP
AXI4-Stream	◦ Data-only burst	Local Link, DSP, TRN, FSL

Table 2.1.: AXI4 Feature Availability and IP Replacement[42]

2.4. Basis for Evaluation

In the thesis, for the evaluation of AXI Interfaces, hardware and software designs are provided.

These designs are implemented on Digilent ZYBO board. ZYBO integrates Zynq Z-7010 SoC device and various peripherals such as Ethernet, USB, SD (Secure Digital) card interface, on-board memories, etc. Please refer to section 3.3 for an overview of the board.

High level view of the design is illustrated in Figure 2.28. It is detailed as follows[[37], [38]]:

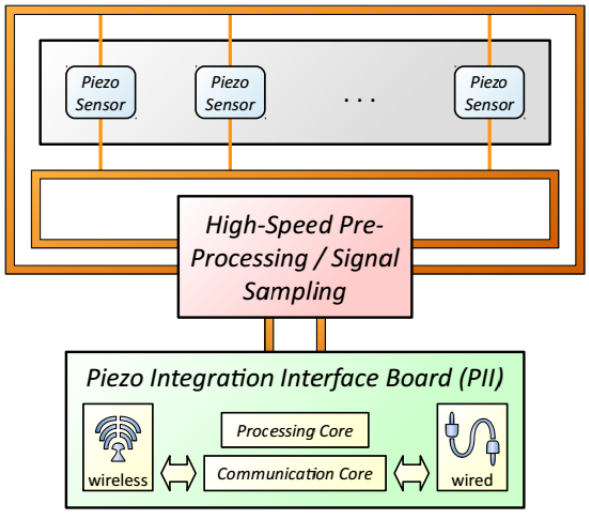


Figure 2.28.: Overview of the design[37]

An electronic circuit is used to pre-process piezo voltage levels before passing it for sampling. This circuit is illustrated in Figure 2.29. It uses a series of Operational Amplifiers (op-amp) to provide useful voltage for XADC module to sample.

First op-amp acts as a buffer to isolate input source from the output so that loading affect on the source are reduced and, in addition, maintains the output voltage level. Second op-amp shifts the input voltage by adding a reference voltage for ADCs with non-symmetric input range. Third op-amp acts as an inverting amplifier to boost the voltage level so that it can be easily sampled. Lastly, fourth op-amp is used to keep the voltage within the limits of ADC input range.

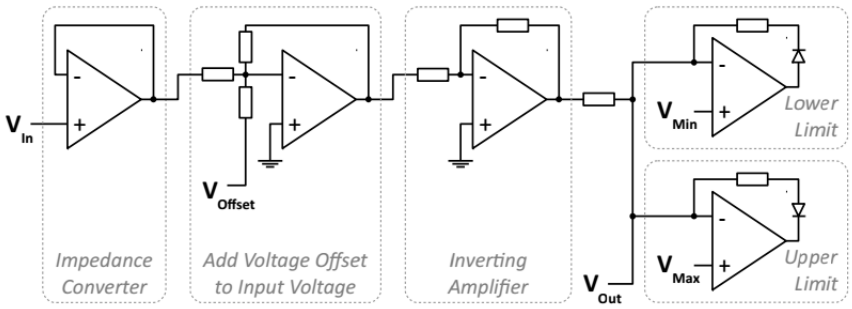


Figure 2.29.: View of Pre-processing Circuit used in the design[38]

Piezo Integeration Interface board makes use of Zynq architecture. It uses PL to sample data via XADC module, then filter unwanted information using a threshold function, and, hence, extract features in terms of voltage-time values. Given FPGA implementation for PL is detailed in section 4.1.

Extracted features are passed to machine learning interface, which is running on PS, to detect and localize the impact.

In summary, basis for the master thesis is ZYBO evaluation board with a given FPGA implementation and a machine learning interface description.

3. Concept

3.1. Comparison of SoC Bus Standards

In section 2.2, various on-chip bus standards were introduced. These includes: ARM AMBA, IBM CoreConnect, Altera Avalon, OpenCores Wishbone, and Accellera Open Core Protocol (OCP). These bus standards are summarized in Table 3.1 in addition with several other standards that are not discussed in the thesis. An overview of comparison between standards, presented in section 2.2, is remarked below.

All standards, except for Avalon, have an open architecture. Avalon has partially open its architecture to public[32]. Among these, only Wishbone is truly freely, provided by OpenCores for public use. Other standards require registration or license agreement. OCP is openly licensed.

All the bus standards implement a synchronous bus system, that is, each communication in the bus is with reference to the bus clock.

Both AMBA and CoreConnect use hierarchical bus architecture, though difference lies in the number of layers. AMBA employs two bus layer: high-performance and low-performance bus, whereas CoreConnect utilizes three bus layer topology: PLB, OPB, and DRC. DRC bus uses ring topology for sharing status and control information. Therefore, CoreConnect makes usage two different topology schemes. Avalon and OCP incorporates similar architecture, point-to-point topology. Wishbone provides multiple choices: Point-to-Point, Ring type, Crossbar network, or Shared bus. It takes a single level bus approach and so does not support for hierarchical buses. For support for multiple buses, two separate Wishbone interfaces can be created: one for high speed, low latency and other for low speed components[27].

Single level bus approach severely limits the bus bandwidth because of increased load on the bus. For this reason, hierarchical bus offers better throughput due to division of high-bandwidth and low-bandwidth components on separate bus lines. Major disadvantage with shared bus system is that it limits design scalability and delay in sharing resources between SoC components[27].

AMBA and Wishbone offers flexibility in arbitration; they do not define any arbitration scheme. It is up to the user to utilize arbitration method according to the application's needs. CoreConnect uses static priority in a programmed fashion. Avalon connects masters and slaves in a point-to-point manner, making no use of bus for the transaction. Therefore, they use slave-side arbitration scheme in case multiple masters intend to access the same slave. OCP does not make use of arbitration. AMBA and CoreConnect have similar data transfer schemes: handshaking,

	Name	AMBA	Avalon	Core Connect	Wishbone	Silicon Backplane	Core Frame	Marble	PI bus	OCP	VCI	Lotterybus
Topology	Point-to-point	-	×	-	×	-	14*	-	-	×	n/a	-
	Ring	-	-	1*	×	-	-	-	-	-	n/a	-
	Unilevel shared bus	-	-	-	×	-	-	-	×	-	n/a	-
	Hierarchical bus	×	-	1*	-	-	-	×	-	-	n/a	×
	Interconnection network	-	-	-	×	×	-	-	-	-	n/a	-
	Synchronous/Asinhronous	S	S	S	S	S	S	A	S	S	S	S
Arbitration	Static priority	7*	13*	4*	3*	-	3*	×	3*	-	3*	-
	TDMA	7*	13*	-	3*	6*	3*	-	3*	-	3*	-
	Lottery	7*	13*	-	3*	-	3*	-	3*	-	3*	×
	Round-robin	7*	13*	-	3*	6*	3*	-	3*	-	3*	-
	Token Passing	7*	13*	-	3*	-	3*	-	3*	-	3*	-
	CDMA	7*	13*	-	3*	-	3*	-	3*	-	3*	-
Bus width	Data bus width [bit]	8*	1-128	9*	8,16,32,64	8,16,32,64	n/a	n/a	1-32	n/a	n/a	n/a
	Address bus width [bit]	32	1-32	10*	1-64	n/a	n/a	n/a	1-32	n/a	n/a	n/a
Transvers	Handshaking	×	-	×	×	×	2*	×	×	×	×	n/a
	Split transfer	×	-	×	n/a	×	-	×	-	×	×	n/a
	Pipelined transfer	×	×	×	-	×	n/a	×	×	×	×	n/a
	Burst transfer	×	×	×	×	×	×	×	-	×	×	×
	Broadcast	-	-	n/a	n/a	×	×	×	-	×	n/a	n/a
	Multicast	-	n/a	n/a	n/a	×	n/a	n/a	-	-	n/a	n/a
	Operating frequancy	11*	n/a	12*	11*	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 3.1.: Features of SoC Bus Standards[27]. Exceptions:1* Data lines shared, control lines point-to-point ring; 2* Palmbus uses handshaking, Mbus does not; 3* Application specific, arbiter can be designed regarding to the application requirements; 4* Programmable priority fairness; 5* Two level arbitration, first level TDMA, second level static priority; 6* Two level arbitration, first TDMA, second round-robin token passing; 7* Application specific except for APB which requires no arbitration; 8* For AHB and ASB bus width is 32, 64, 128 or 256 byte, for APB 8, 16 or 32 byte; 9* For PLB bus width is 32, 64, 128 or 256 byte, for OPB 8, 16 or 32 byte and for DCR 32 byte; 10* For PLB and OPB bus width is 32 byte, and for DCR 10 byte; 11* User defined operating frequency; 12* Operating frequency depending on PLB width; 13* Slave side arbitration; 14* System of buses, Palmbus and Mbus, both are point-to-point.

pipelining, split, and burst transfers. Wishbone supports handshaking and burst transfers. It can support pipelining if the architecture utilizes data flow interconnection scheme. Avalon makes no use of handshaking or split transactions since transfers take place without the intervention of bus. It employs pipelined and burst transfers. OCP does not support split transfers, however, it is the only one among others that provides broadcast transfers.

Among these, Wishbone is the only standard that offers read-modify-write transfer.

In conclusion, CoreConnect has similar feature set as AMBA. Compared to AMBA, Wishbone provides simpler design and better flexibility. However, Wishbone allows maximum data width of 64 bits whereas AMBA can provide data width of up to 256 bits. As mentioned in section 3.2, one of the evaluation criteria is the amount of data transferred between FPGA and processor. Hence, it is useful to have support for higher data widths. In terms of operating frequency, AMBA and Wishbone provide flexibility to user whereas CoreConnect's frequeny is limited to Processor Local Bus (PLB) width.

3.2. Approach for Evaluation of AXI Interfaces

The goal of the master thesis is to analyze best fitting AXI Interface for the given hardware and software design. AXI4 Interfaces have to be evaluated with different configurations resulting in quantification of best fitting AXI4 Interface. Hence, to satisfy such needs, an approach is outlined as follows.

1. Understanding of given FPGA implementation to derive the changes needed in the design to incorporate best-fitting interface.
2. Analysis of three different AXI4 Interfaces:
   2.1. Hardware design for three AXI4 Interfaces
   2.2. Implementation of several test cases to evaluate performance of each AXI4 Interface hardware design.
3. Based on test results, conclusion derived about the efficient usage of each AXI4 protocol.

Criteria for evaluation of the interfaces includes:

- Amount of data (input length) transferred between PS and PL
- Processing time for read/write operations
- PL Frequency
- Various interface configuration such as AXI4-Full provides different data widths and several burst configuration

The influence of PL frequency on the performance of read/write operation is to be included in the results. Moreover, AXI4-Full protocol provides several configuration for data widths and burst transfer. Performance of AXI4-Full Interface is to be evaluated with these configurations.

3.3. Evaluation Platform

3.3.1. Hardware

Digilent ZYBO board

Digilent provides development platform for SoC (Figure 3.1) that features Xilinx Zynq Z-7010 All Programmable System-on-Chip (APSoC) architecture. Zynq architecure is discussed in section 2.3.

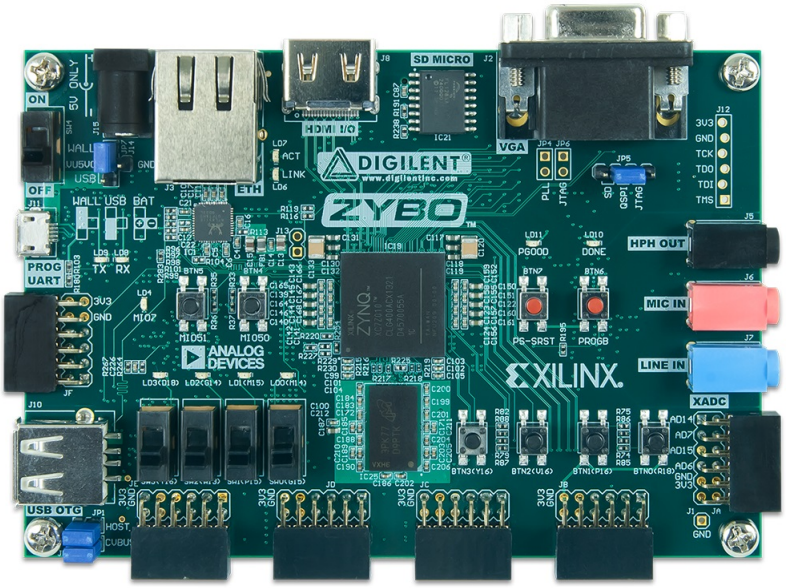


Figure 3.1.: ZYBO ZYNQ-7000 Development Board[13]

High-level view of the architecture can be seen in Figure 2.23. Some of the main features offered by ZYBO are listed as follows[13]:

- 650 MHz dual-core ARM Cortex-A9 processor
- Xilinx Artix-7 FPGA Programmable Logic
- 512 MB DDR3 memory

Functionality for On-board JTAG programming is featured by the board. It also includes dual analog/digital Pmod (Peripheral Module) interface. Four channels of

this interface are used in the MERGE project to connect ADC signals from piezo electrodes placed on the metal sheet. Various other interfaces are also supplied such as 1G Ethernet, USB 2.0, SPI, UART, CAN, I2C, SDIO, HDMI, etc.

ZYBO supports three different boot modes: microSD, Quad-SPI (QSPI) Flash, and JTAG. In the thesis, Linux based OS is used which is stored in microSD. ZYBO is configured to load Linux kernel from SD card into RAM via JP5 jumper[13].

Figure 3.2 highlights clock usage for PS and PL. ZYBO provides 50 MHz clock to PS. This allows processor to run at maximum frequency of 650 MHz and DDR3 memory controller at maximum of 525 MHz[13]. PS incorporates four PLLs (Phase-Locked Loop) depicted as *FCLK_CLK[0:3]*. They provide reference clocks to PL. In addition, 125 MHz clock from Ethernet module provides an external clock input to PL.

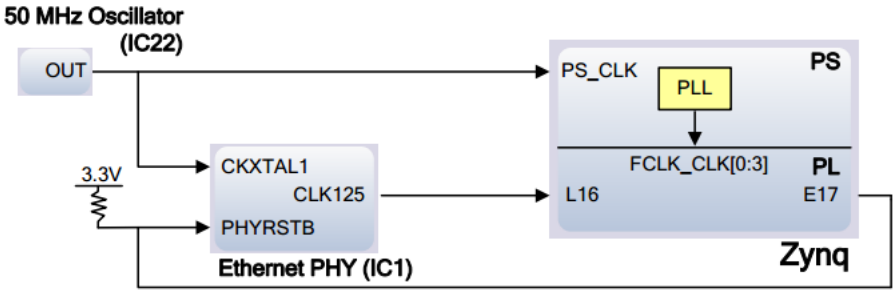


Figure 3.2.: ZYBO clocking scheme[13]

ZYBO provides four pairs for analog input channels. In the given FPGA implementation, XADC module in PL reads piezo voltages through these analog channels.

3.3.2. Software

For SoC devices with FPGA, Xilinx offers Vivado Design Suite and Xilinx Software Development Kit (SDK). These tools have simplified embedded system design by providing a user-friendly interface, debug support, and verification of design.

In the thesis, Vivado v2015.4.2 is used to design hardware for different AXI4-Interfaces. To test the hardware design, several test cases were implemented in C using SDK.

Vivado

Xilinx provides Vivado IDE (Integerated Design Environment) as a graphical user interface for the Vivado Design Suit. It eases embedded hardware designing by providing several built-in IP blocks such as Zynq Processing System, AXI Interconnect, AXI DMA, AXI4-Stream Data FIFO, etc. User simply has to drag-and-drop IP blocks and interconnect them graphically.

Design flow for Vivado is depicted in Figure 3.3. IP Catalog lists options to add IP to the design. Vivado offers interaction at different design stages: Register-Transfer-Level (RTL) elaboration, synthesis, and implementation[46]. In contrast to Xilinx ISE (Integerated Synthesis Environment), Vivado incorporates built-in simulation techniques.

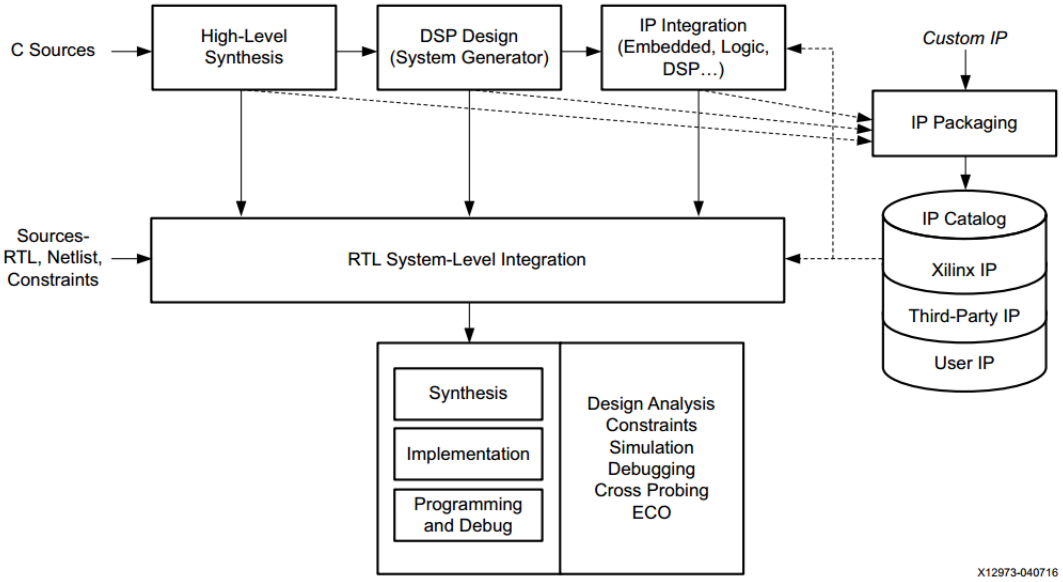


Figure 3.3.: High-Level Design Flow in Vivado[46]

In Programming and Debug stage, Vivado generates a bitstream file that encodes information of implemented hardware design as binary data. PS loads this file to program the PL.

SDK

Xilinx SDK is based on Eclipse open standard. It provides a design environment for building embedded applications to run on PS. Some of the feautres offered by SDK are mentioned as follows[40]:

It provides a GNU-based compiler toolchain with various build configuration for the application. Most importantly, it features a rich set of C/C++ development environment.

SDK offers automatic Makefile generation. Multiple debugging options are available for embedded targets. It also have an option for FPGA and Flash programming. For Xilinx built-in IP cores, it supplies driver support with documentation and example codes.

Design flow in Xilinx SDK is represented in Figure 3.4. Software application source and header files are compiled using gcc/g++ compiler. It generates an object code file which represents the application in binary form.

Required system libraries for the application is provided by software platform. These libraries and object code is linked together via linker script. Finally, an Executable and Linkable Format (ELF) file is generated. This ELF file is transferred to the board so that PS can execute the software application.

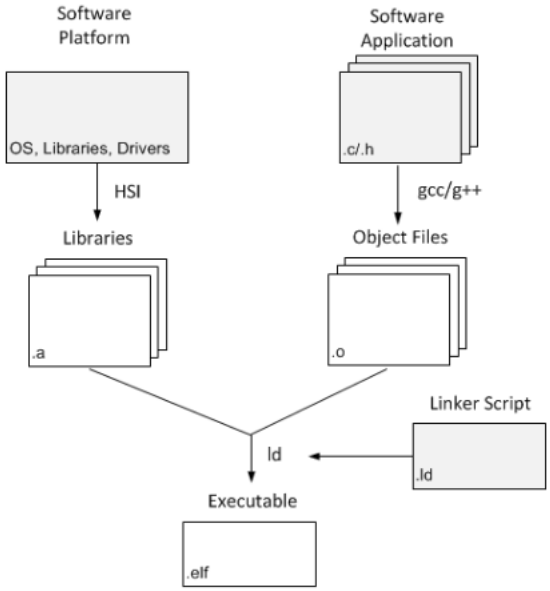


Figure 3.4.: Software Workflow in SDK[40]

4. Implementation

4.1. Overview of FPGA Implementation

Top-level view of the provided hardware design is illustrated in Figure 4.1. In this design, *MERGE_PL* IP is the given implemented design for PL. It performs analog to digital conversion (ADC) of signals obtained from piezo composite. With this digital data, appropriate features are extracted for the machine learning interface.

To achieve real-time detection of impact on a piezo metal composite, it requires a frequent delivery of feature data from PL. Unlike memory-mapped interfaces, AXI4-Stream protocol does not require address information for the transaction. This reduces the amount of address/control signals needed for the transfer and enables unlimited burst length. Therefore, the current design makes use of AXI4-Stream interface to transmit features from *MERGE_PL* IP to PS via AXI DMA controller.

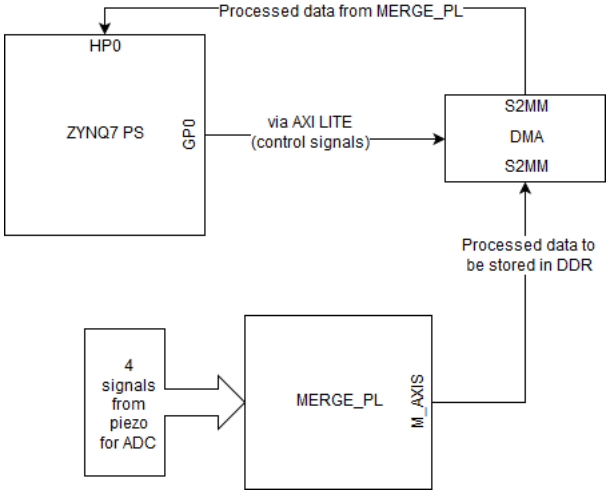


Figure 4.1.: Block Diagram of Hardware Design

ZYNQ7 PS IP provides configuration settings for clock, I/O pins, DDR memory, interrupts between PS and PL, etc. This IP provides High-Performance port (HP0) which is required to allow high throughput datapath between *MERGE_PL* master and DDR slave. This port features FIFO buffers to increase performance of data flow. To communicate directly with AXI DMA IP, PS use a low-performance General Purpose (GP) port 0. It is suitable for transmitting control signals as highlighted in Figure 4.1.

`AXI DMA` IP is used to provide support for *MERGE_PL* IP to communicate with DDR memory. For communication, it offers two channels: MM2S (Memory-Mapped to Stream), for reading data from memory, and S2MM (Stream to Memory-Mapped), for writing data to the memory. For the two channels, it supplies various configurations such as flexible address/data width and burst size alternatives.

To make necessary changes in the provided design so that best fitting AXI4-Interface can be incorporated, it requires an understanding of the *MERGE_PL* IP. Overview of its hardware implementation is given as follows.

4.1.1. MERGE_PL FPGA IP

This IP is responsible for signal processing of four analog input channels and extraction of appropriate features, from the sampled values, for the machine learning interface. Hierarchical layout of the design is represented in Figure 4.3.

Input signals necessary for the design consists of: clock which is provided by PS and reset that is obtained from Processor System Reset IP. This IP is responsible for generating synchronous reset for interconnects and peripherals.

Other input signals used to provide functionality for *MERGE_PL* are analog differential and AXI4-Stream interface signals. These analog voltage levels are extracted from piezo sensors which are placed on a metal sheet. These voltage levels acts as an input for ADC. *tReady* signal is needed by master to know if slave is ready for the current transaction. AXI4-Stream uses this signal to facilitate two-way handshake mechanism as was shown in Figure 2.26.

If *tReady* signal is HIGH, then master can initiate the transaction by driving an output signal,*tValid*, to HIGH. Other output signals used in the IP are AXI4-Stream based signals: *tData*, *tlast* and *tStrb*.

tData carries 32-bit feature data shown in Figure 4.2. *tlast* signal indicates the last data transfer in the stream. *tStrb* signal provides option for selecting valid data lines in the stream. For example, if for a 32-bit channel, data to be transferred is less than 32-bits, then *tStrb* signifies appropriate byte lanes (4 byte lanes for 32-bits) to use that carry necessary information. In the provided design, all byte lanes are high for 32-bit feature data.

As shown in Figure 4.2, feature data includes information about current analog channel and feature value associated to it. *Time_Channel* stores the feature value obtained from corresponding analog channel. *Time_ID* is used as an identifier to correlate the stored value with correct analog channel.

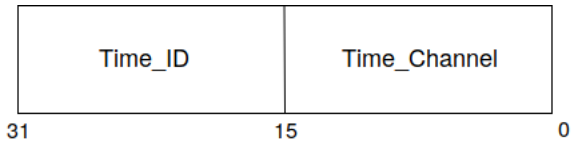


Figure 4.2.: Feature Data Frame

From Figure 4.3, we see that the design makes use of three main modules: *MERGE_PL_v0_1*, *MERGE_PL_v0_1_M00_AXIS*, and *analog2digital*.

analog2digital module converts analog voltage level to digital value and uses sub-modules to extract suitable amount of features. As shown in Figure 4.3, it contains three sub-modules: *moving_average*, *Sliding_window_sum_integerator*, and *timediff_extractor*. *moving_average* calculates the average value of sampled data stored in FIFO (First-In-First-Out) and passes it to *Sliding_window_sum_integerator*. *Sliding_window_sum_integerator* add these values and store them in a FIFO to account for previous voltage levels. *timediff_extractor* extracts sufficient amount of features from the values provided by *Sliding_window_sum_integerator*.

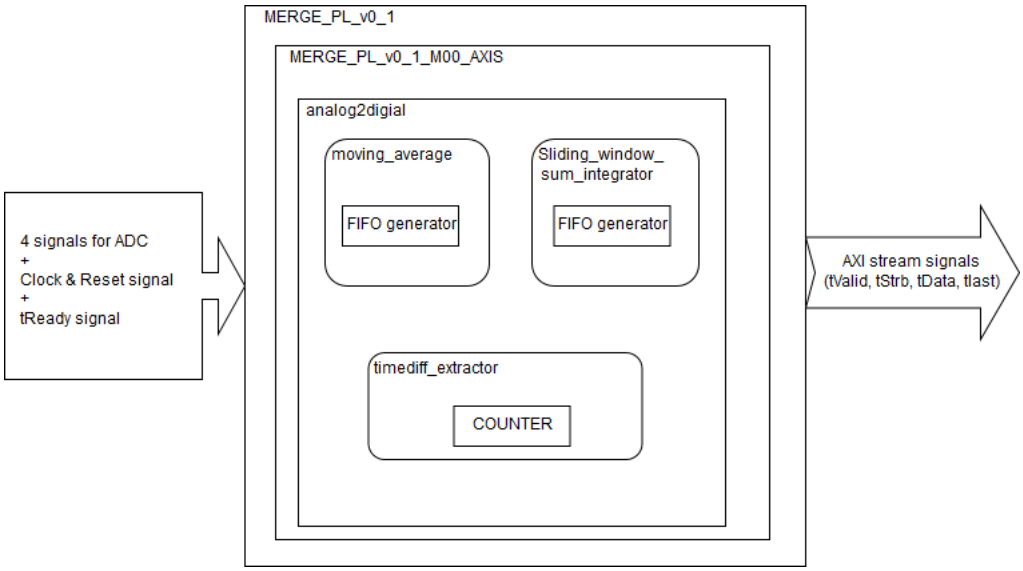


Figure 4.3.: Block Diagram of MERGE_PL Implementation

This extracted data is passed to *MERGE_PL_v0_1_M00_AXIS* module which puts the data on the stream. *MERGE_PL_v0_1* is the top module that bundles the functionality and provides an interface for input/output signals. Sub-modules of *MERGE_PL_v0_1* are discussed in detail as follows.

MERGE_PL_v0_1_M00_AXIS

This module passes analog differential signals to *analog2digital*. In addition, it is responsible for reading feature extracted data from timediff_extractor in *analog2digital* module and passing it to DMA via AXI4-Stream interface. A sketch of state machine used for this purpose is depicted in Figure 4.4.

Flow of the state machine is discussed as follows:

1. State machine starts in the idle state (ST_IDLE) and remains in this state until a valid feature data is received from *timediff_extractor*. A HIGH state of *time_valid* depicted in Figure 4.4 represents a valid data. *save_data* is used as

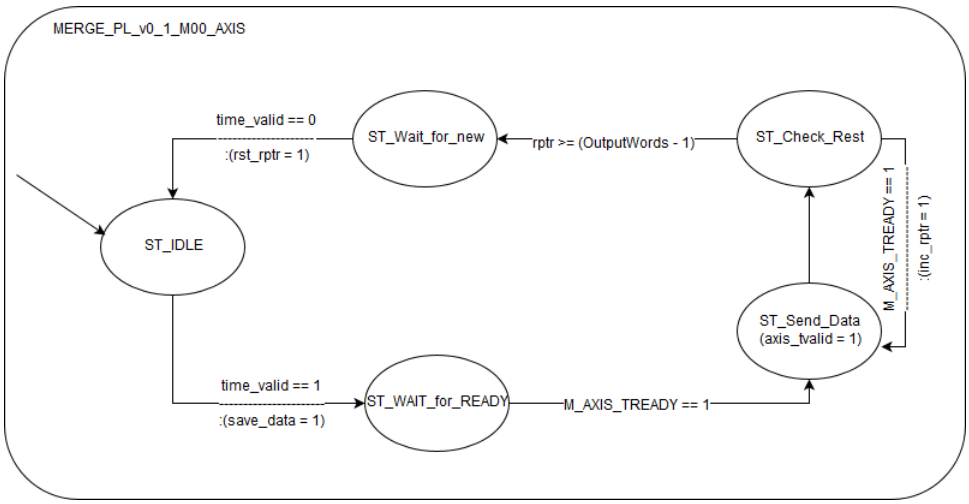


Figure 4.4.: State Machine Diagram. Statement above the arrow denotes a condition. Statements below the arrow denotes statement executed if corresponding condition satisfies.

a flag to store this data into a register. In such condition, control of the state machine is transferred to ST_WAIT_for_READY state.

2. Control remains in ST_WAIT_for_READY state until slave indicates that it is ready to accept data. Slave drives *M_AXIS_TREADY* signal HIGH for data acceptance and thus control is passed to ST_Send_Data state.

3. In ST_Send_Data state, master drives a valid signal (*axis_tvalid*) HIGH, thereby, puts data on the stream. As the data is transferred to stream, control is passed to ST_Check_Rest state which checks for two conditions: amount of data left to be transferred and ready signal from slave. If there is data left to be transferred and slave is ready to accept it, then control is given back to ST_Send_Data state. `inc_rptr` is used as a flag to increment *rptr* which keeps a count of the number of data on the stream. In total, four analog channels are used in the design. Therefore, four 32-bit data values have to be transferred to slave. *OutputWords* shown in Figure 4.4 stores the number of analog channels. When all data values are transferred, *inc_rptr* increments *rptr* to a value greater than *OutputWords*. This results in transfer of control to ST_Wait_for_New state.

4. In this state, control checks for *time_valid* signal to know if data is invalid. As the *time_valid* switches to LOW, *rptr* is set to zero by *rst_rptr* flag. Thus, the control is transferred back to ST_IDLE state where it waits for next set of feature data.

When *TREADY* signal is HIGH, data is sent across the AXI4-Stream interface until data from all analog channels is transmitted. *OutputWords* stores the number

of analog input channels used. *time_valid* signal is driven by timediff_extractor to indicate the availability of feature data.

Analog2digital

This is the core module that converts analog differential signals from piezo sensors to a corresponding digital value. As shown in Figure 4.5, it receives these analog signals from four analog channels (06, 07, 14, and 15) provided in ZYBO.

Whenever there is an impact on a metal sheet, a mechanical wave travels across the material. Piezo sensors near to the source of wave register a higher voltage level. Whereas, at the same time, sensors far from the point of impact will have a lower or different voltage level. Therefore, *timediff_extractor* makes use of such time/voltage correspondence to locate the point of impact[38].

Sampled voltage levels are passed through `moving_average` and *Sliding_window_sum_integerator* modules. Former module acts as threshold for forwarding voltage levels that are relevant for the detection of impact[38]. This helps in filtering of unnecessary information.

Sliding_window_sum_integerator records the previous average voltage levels to notice when voltage levels starts to rise[38]. FIFO (First-In-First-Out) generator shown in the Figure 4.5 is used to hold required amount of data for calculating average and integration of values for *timediff_extractor*.

These steps are highlighted in Figure 4.5.

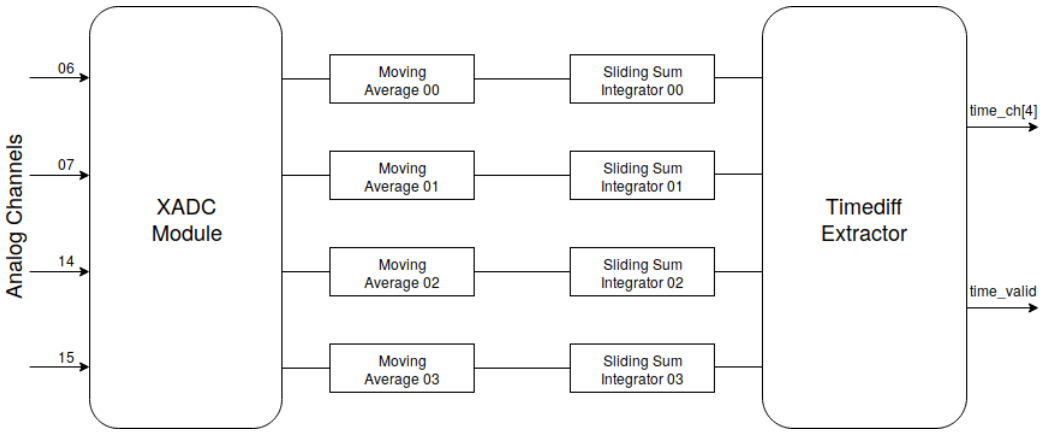


Figure 4.5.: Block Diagram of analog2digital Module

Four analog channels (06, 07, 14, 15) requires its own moving average and sliding sum integrator. XADC provides two ADCs (ADC A and ADC B) to enable simultaneous sampling mode[49]. ADC channel 6 and 14 utilize ADC A and ADC B respectively. Similarly, ADC A for channel 7 and ADC B for channel 15. These ADC channels are assigned specific addresses through which XADC module selects them. In the following Table 4.1, value of ADC addresses are listed.

ADC Channel	Memory Address
06	0x16 (22d)
07	0x17 (23d)
14	0x1E (30d)
15	0x1F (31d)

Table 4.1.: Addresses of ADC Channel Registers[49]

Usage of XADC module is highlighted using a state machine in Figure 4.6. The flow of the state diagram is discussed below:

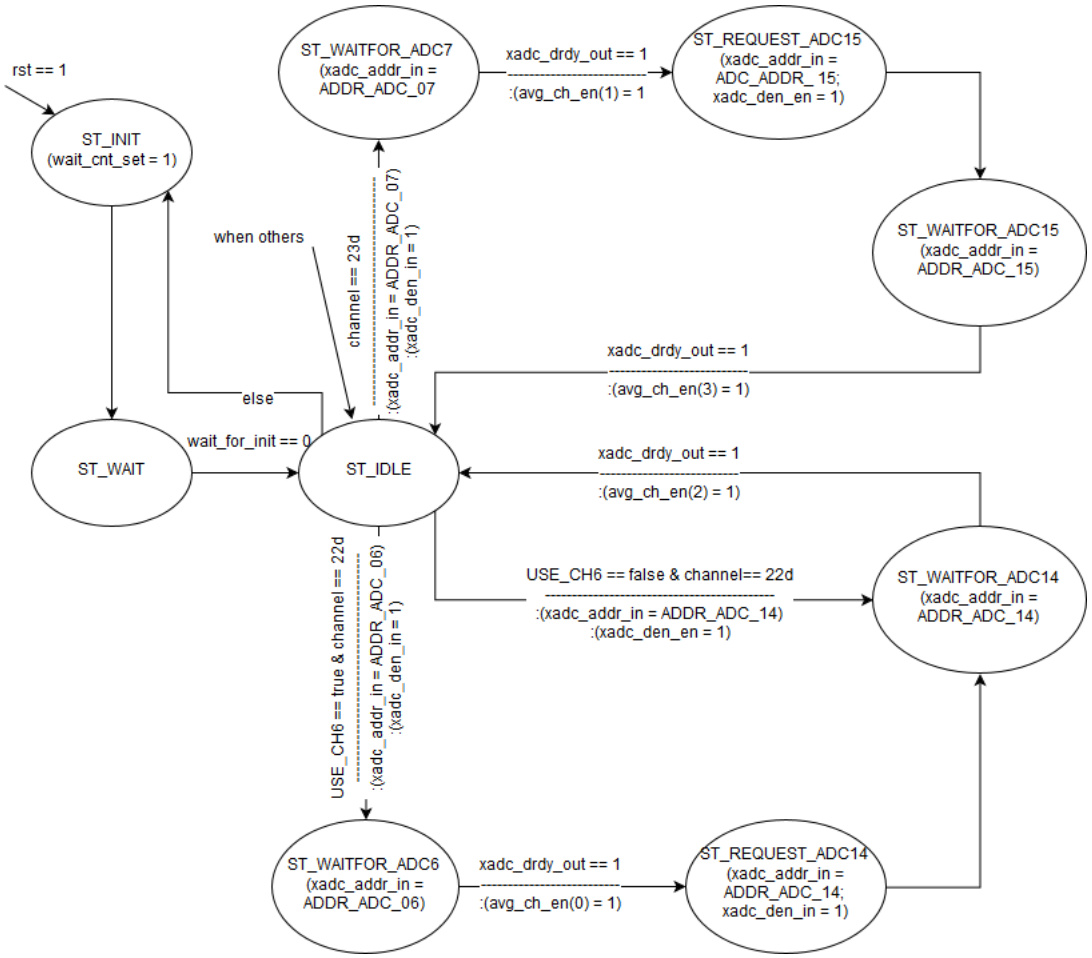


Figure 4.6.: State Machine Diagram. Statement above the arrow denotes a condition. Statements below the arrow denotes statement executed if corresponding condition satisfies.

1. As the system resets, state machine begins from ST_INIT state. Control passes to ST_WAIT state. In ST_WAIT state, control is passed to ST_IDLE state when *wait_for_init* is LOW. *wait_for_init* and *wait_cnt_set* signals are used to provide initialization time for XADC.

2. ST_IDLE is the main state responsible for polling XADC module. *xadc_addr_in* stores an address of ADC channel and *xadc_den_in* signals that current ADC channel can be read if the signal is HIGH. ADDR_ADC_(6/7/14/15) are constant value which stores the addresses of ADC channel registers previously mentioned in Table 4.1. USE_CH6 is a boolean constant that specifies if ADC channel 06 is to be used for conversion or skipped.

   2.1. If USE_CH6 is TRUE and selected channel address is 22d (ADC 06), then *xadc_addr_in* stores the address of ADC channel 6. It is enabled to be read for voltage level by *xadc_den_in*. Additionally, control is transferred to ST_WAITFOR_ADC6 state. In this state, *xadc_addr_in* value is held to ADC 06 address. Whenever XADC samples a analog value, it drives *xadc_drdy_out* output HIGH. When sampled value for ADC 06 is available, then it is passed to moving average 00 module (Figure 4.5) via enable signal (`avg_ch_en(0)`). Simultaneously, the control is passed to ST_REQUEST_ADC14 state.

   In ST_REQUEST_ADC14 state machine requests for sampled value of ADC 14 by giving appropriate address and setting read address enable signal. Then the control is passed to ST_WAITFOR_ADC14 where controller waits for the sampled value. As soon as *xadc_drdy_out* is HIGH, digital value of ADC 14 is passed to moving average 02 module. And thus the control is transferred back to ST_IDLE where state machine waits for next ADC channel to process.

   2.2. If USE_CH6 is FALSE and selected ADC channel is 22d, then it directly requests for sample value of ADC 14 by assigning appropriate address to *xadc_addr_in* and enabling the read address signal. Same as above, after the sampled value is available, control is passed to ST_IDLE state.

   2.3. If sampled value of channel 23d (ADC 07) is required, then control is passed to ST_WAITFOR_ADC7 state. *xadc_addr_in* holds the address of channel 23d (ADDR_ADC_07). As soon as sampled value is obtained, moving average 01 receives the value.

   Access is transferred to next state (ST_REQUEST_ADC15) where sampled value of ADC 15 is requested by providing appropriate address. Afterwards, ST_WAITFOR_ADC15 state holds the address of ADC 15 and receives the sampled value. Like other channels, digital value is passed to moving average 03 and the control is given back to ST_IDLE.

   2.4. If no ADC channel has to be processed, then ST_IDLE provides control to ST_INIT.

4.2. Hardware and Software Design for Evaluation

Hardware is implemented in Vivado tool which enables synthesis and generation of bitstream for hardware designs. As mentioned in section 3.3, it provides necessary framework to facilitate the design by including pre-built IP components. Software is implemented in Xilinx SDK that provides an Integrated Development Environment (IDE) with built-in features for compilation and configuration of the implemented code.

To evaluate the performance of different AXI4-Interfaces, implementation of hardware and software done in the thesis are discussed in the following sections.

4.2.1. Hardware Design

AXI4-Lite Interface

Figure 4.7 depicts implemented design for AXI4-Lite Interface. *Slave_lite_300ip* (Lite IP) is custom AXI4-Lite IP that will be evaluated for its performance. Such custom is created using 'Create and Package IP' feature provided by Vivado. User can select the interface type and various other configuration such as: mode of interface (master/slave), data width, memory size, and number of registers. 'Create and Package IP' produces an AXI protocol design in HDL (Hardware Description Language) such that custom IP can be used directly for basic transactions.

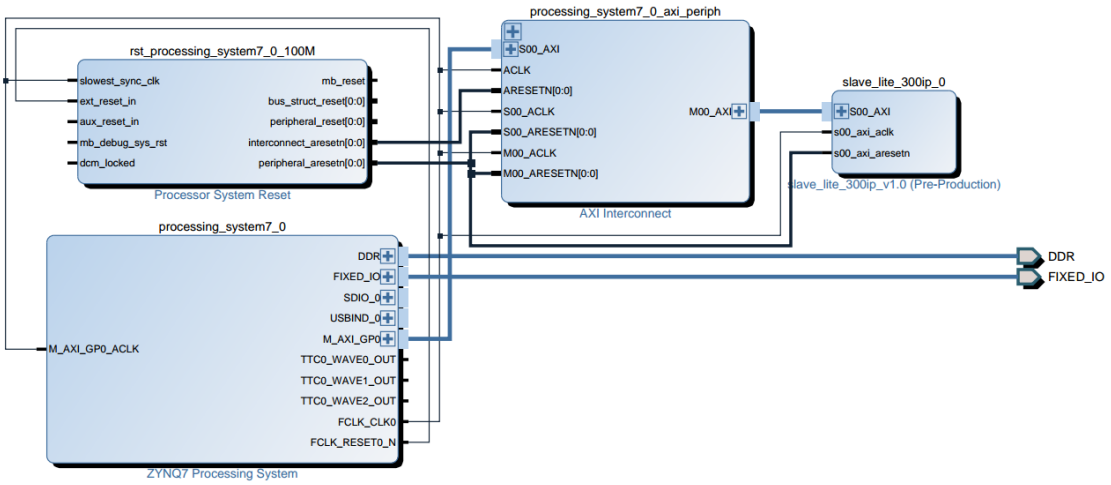


Figure 4.7.: Hardware Design for AXI4-Lite Interface

Lite IP features 300 slave registers, each of size 32-bits. These registers acts as memory interface for Lite IP to allow PS to perform read/write operation on them. These slave registers are mapped into address space provided for GP0. Vivado provides user an option for filling addresses for IPs in PL via address editor. Figure 4.8 highlights the address (0x43C0_0000) of Lite IP used in the design. Address can be manipulated to a value that supprots GP0 address space.

Figure 2.24 in chapter 3 provides a memory layout for ZYNQ7 PS. Therefore, PS uses respective mapped address for slave register to do read/write operation on it.

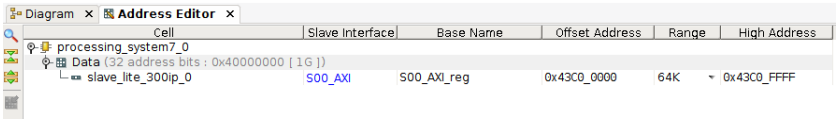


Figure 4.8.: View of Address Editor

Clock and reset signals are common for all IPs in PL. PL clock is provided by PS shown in ZYNQ7 PS IP as *FCLK_CLK0*. Reset signals for peripherals and interconnect are produced by Processor System Rest IP. This IP is responsible for generating synchronous resets. It is seen from Figure 4.7 that resets are in sync with *slowest_sync_clk* (connected to *FCLK_CLK0*). PS resets this IP using *FCLK_RESET0_N*.

Data flow between PS and Lite IP is enabled by AXI Interconnect. AXI Interconnect features several options for communication interface such as: multiple master/slave interfaces, data FIFO, and register slices[48]. It allows mixture of AXI master and slave devices to communicate with each other with different interfaces or data widths.

Each AXI Inteconnect makes use of AXI Crossbar. AXI Crossbar is an IP core used to connect similar memory-mapped masters and slaves. GP0 supports AXI3 protocol whereas *Slave_lite_300ip* interface uses AXI4 protocol. Therefore, in the current design, interconnect makes use of AXI Protocol Converter shown in Figure 4.9. This IP core provides feature for interfacing different AXI based memory-mapped protocol.

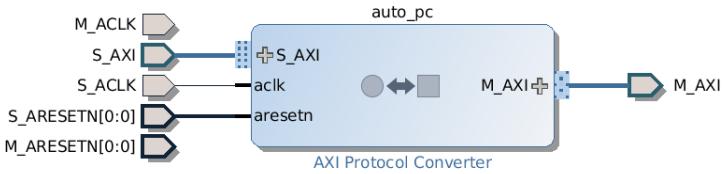


Figure 4.9.: AXI Protocol Converter IP core

In Figure 4.9, *S_AXI* is connected to GP0 master and *M_AXI* is interfaced to the slave interface (*S00_AXI* of Lite IP. It requires clock and reset signals for both master and slave to perform appropriate conversion without loss of transfer data.

Data, which is transferred between PS and Lite IP, consists of 32-bit integer value. For write operation, this value is written to a slave register. Similarly, for read operation, an integer value present in a slave register is passed to software design running on PS.

AXI4-Full Interface

For evaluation of AXI4-Full, the hardware design used in the thesis is depicted in Figure 4.10. In the design, *Slave_full1024bytes_ip* (Full IP) is concerned IP which provides testing for AXI4-Full interface.

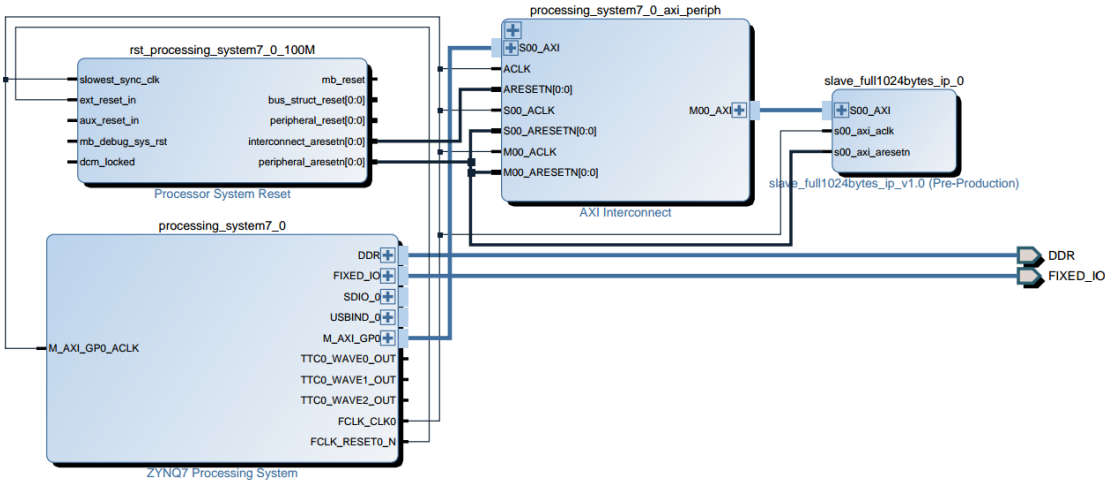


Figure 4.10.: Hardware Design for AXI4-Full Interface

Full IP provides 1024 bytes of memory interface for read/write operations by PS. Memory size is set while creating AXI Full custom IP in 'Create and Package IP' tool. In the tool, Vivado provides various memory sizes listed as: 64, 128, 256, and 1024. Similar to Lite IP, Full IP uses 32-bit data width. It provides an address width of 10-bit for accessing 1024 bytes of memory.

Vivado provides a feature to customize parameters of AXI4 custom IP after it has been created through Create and Package' tool. For example, customization parameters for Full IP is shown in Figure 4.11 where it presents various parameters for AXI4-Full interface.

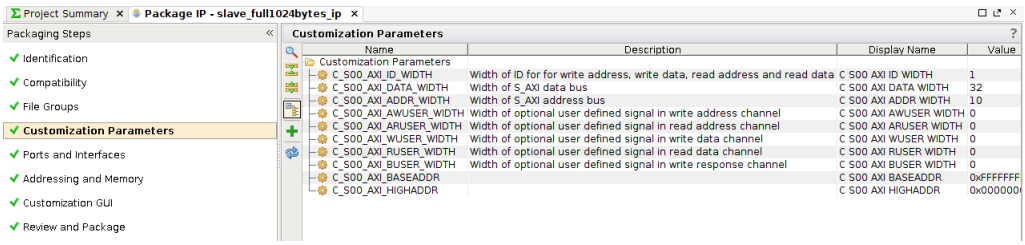


Figure 4.11.: Customization of Parameters in IP Packager

For instance, to check the performance of Full IP based on different data width configuration, customization parameter *C_S00_AXI_DATA_WIDTH* can be changed to respective value. However, if slave uses data width wider than supported by the port in PS, then Vivado uses data width converter IP in the design (Figure 4.12).

Interconnect features AXI data width converter IP core for connecting AXI master and slave having a wider or narrower data width[48].

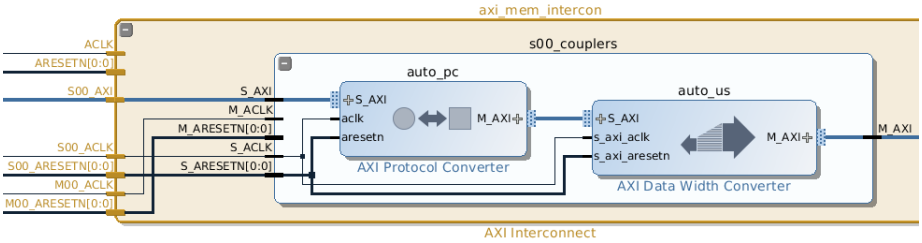


Figure 4.12.: Interconnect Connection with AXI Data Width Converter IP

Full IP is connected to PS via interconnect which, as mentioned before, provides necessary protocol conversion for AXI3 and AXI4 as depicted in Figure 4.12. For Full IP, memory address for Full IP is set to 0x7AA00000 in address editor. PS uses this address to communicate with 1024 bytes of memory.

Similar to Lite IP, Full IP uses a 32-bit integer data which is to be transferred across slave memory. Apart from Full IP, other components added to design serves the same purpose. ZYNQ7 PS uses GP0 port to communicate with Full IP via interconnect. It provides PL clock (*FCLK_CLK0*) to Processor System Reset IP. This IP uses this clock to supply synchronous reset signals for the design.

AXI4-Stream Interface

To test the performance of AXI4-Stream Interface, a block design for the hardware is shown in Figure 4.13.

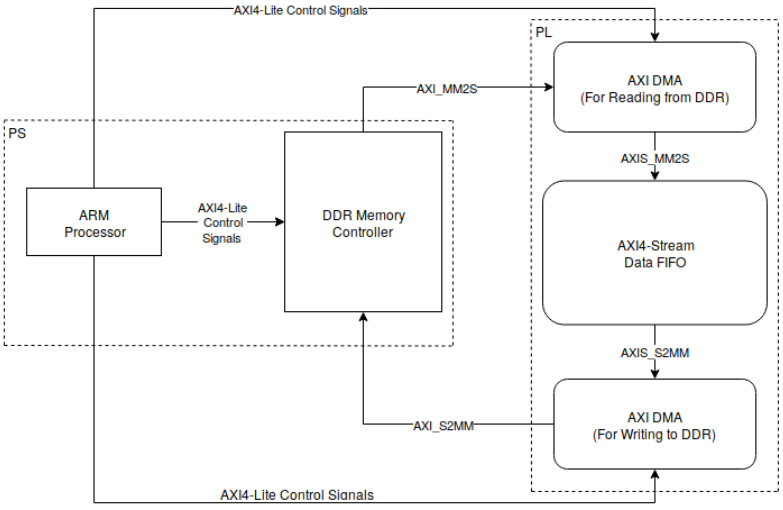


Figure 4.13.: Block Diagram of AXI4-Stream Hardware Design

For a complete view of the hardware, Figure 4.21 is illustrated near the end of this section. The block diagram represents main components used for the flow of data

for AXI Stream interface. It also highlights placement of appropriate components in PS and PL.

To give a general outlook of the data flow, two AXI DMAs are included for read and write operation respectively. AXI4-Stream Data FIFO IP provides buffer feature for the stream data. For read or write operation, DMA provides a set of signals for read (MM2S) and write channel(S2MM). These set of signals provide data flow between memory-mapped and stream interface.

For instance, from Figure 4.13 we see that, for MM2S channel, *AXI_MM2S* signal carries data over memory-mapped interface and *AXIS_MM2S* signal transfers data over stream interface.

As seen in Figure 4.13, ARM processor uses AXI4-Lite interface to send/receive control information from DDR memory controller and two DMAs. Overview of some of the main components used in the design are discussed below.

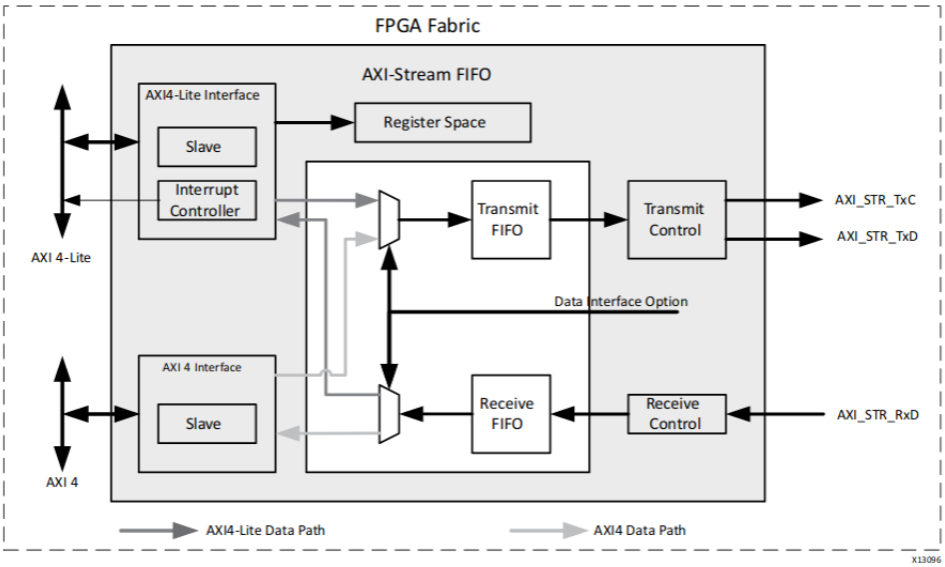


Figure 4.14.: Block Diagram of AXI4-Stream FIFO core[45]

AXI4-Stream Data FIFO IP, provided by Vivado, features communication between memory-mapped and stream interfaces. This IP provides read or write of data packets across a device without the complexity of AXI4-Stream signals[45]. Major components used by this IP are shown in Figure 4.14.

Block diagram represents AXI4/AXI4-Lite interface components to provide communication between them and AXI4-Stream interface. FIFO buffers are installed separately for read and transmit data. Signals: *TxC, TxD, RxD*, shown in Figure 4.14 provides support for data flow between memory-mapped to an AXI4-Stream interface connected to other IP, such as the AXI Ethernet core[45].

From the block diagram of provided hardware design (Figure 4.1), AXI DMA is used to transfer stream data from *MERGE_PL* to PS. So, to test the performance of stream data between PL-PS via AXI DMA, in place of *MERGE_PL*, a buffer is needed. Such that it can store data when PS requests to write data to the stream

via AXI DMA. Similarly, it can provide correct data to PS when requested to read data off the stream.

Therefore, to test the performance of AXI4-Stream interface over AXI DMA Controller, AXI4-Stream Data FIFO IP is used as a storage device in Figure 4.13.

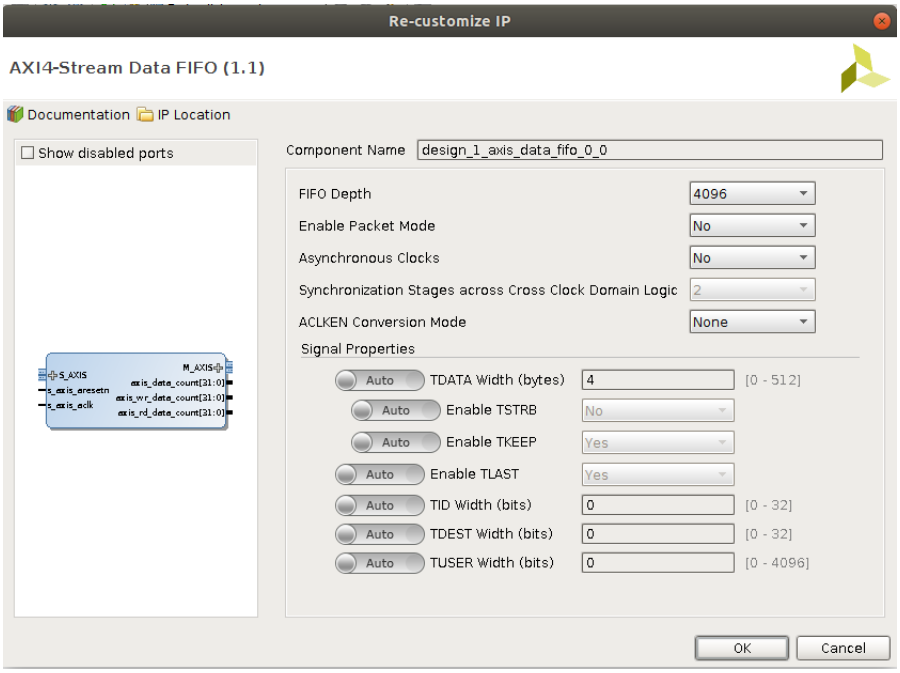


Figure 4.15.: Customization Options for AXI4-Stream Data FIFO

AXI Stream Data FIFO IP incorporated in the hardware design (Figure 4.21) uses a FIFO depth of 4096. This IP provides range of FIFO depth from 16 to 32768. It also offers customization of several other parameters as depicted in Figure 4.15. It shows the IP settings used for the hardware design. Currently, AXI4-Stream signal properties are set to auto mode but can be customized according to the requirement.

As mentioned before in section 4.1, AXI DMA provides set of configurable options for both read and write channels. Configuration used in the hardware design are highlighted in Figure 4.16. Both channels supports maximum burst size of 256 and stream data width of 32-bits. DMA IP provides configuration for memory mapped data width from 32 to 1024 bits. In Figure 4.13, 32-bits is used.

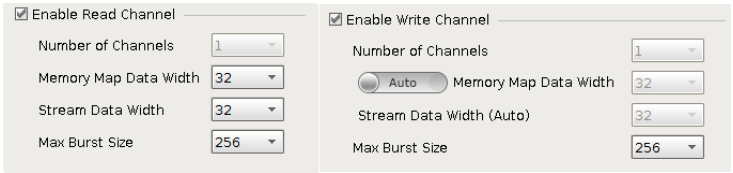


Figure 4.16.: Settings for Read and Write Channel

As seen from Figure 4.21, these DMA controllers are connected to High-Performance Port 0 (HP0) via AXI Interconnect. HP0 provides high-bandwidth

connection of DMA to DDR memory controller. These ports are also called AFI (AXI FIFO Interface) since each interface includes two FIFO buffers for read and write data stream[52].

Each DMA controller is assigned an address in system memory. In address editor, *read_dma* and *write_dma* are assigned to memory address of 0x40410000 and 0x40400000 respectively.

Advantage of using DMA is that it keeps the processor free by overlooking read-/write access to DDR memory. So, processor can do other task while the data flow is being managed by DMA. Hence, DMA needs to inform PS as soon as the data transfer is complete. Figure 4.17 highlights connection of *mm2s_introut* for read channel and *s2mm_introut* for write channel to ZYNQ7 PS.

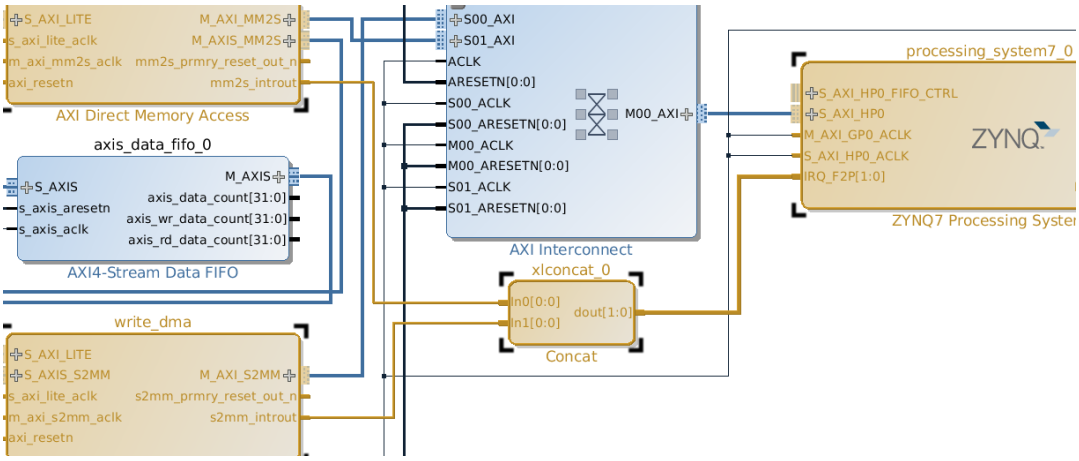


Figure 4.17.: Connection of DMA Interrupts

Vivado provides Concat IP core to combine multiple signals on a different bus into a single bus. In Figure 4.17, it combines two interrupt signals into a single bus of size 2-bits (*dout[2]*.

ZYNQ7 PS provides selection of 16-bit shared PL-PS interrupt port. Since, in this case, PS needs to receive interrupt from two PL components, *IRQ_F2P* of size 2 bits is used in the design (Figure 4.17).

Frequency Test Design

Performance evaluation of AXI4 memory-mapped interface with respect to PL frequency is tested in the thesis. Figure 4.18 illustrates the hardware design used for evaluating the impact of frequency on AXI4-Lite slave IP. In the design, all components, except clocking wizard, have been discussed in the preceding sections. Clocking wizard version 5.2 IP is used to provide different PL frequencies.

Xilinx provides Clocking Wizard IP to help users fulfill clock requirements in the design. Figure 4.19 illustrates the clocking network architecture used by the wizard. It consists of: input and output clocks, buffers, clocking primitive, and optional feedback for phase alignment.

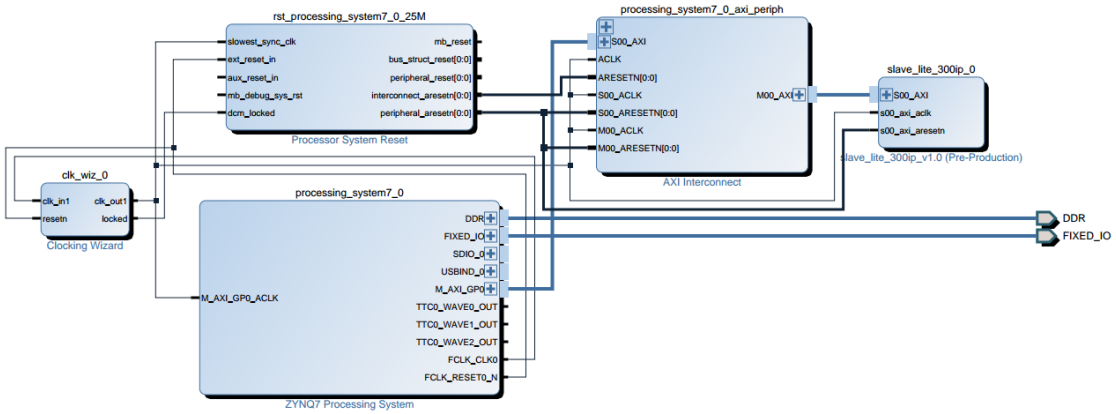


Figure 4.18.: Hardware Design for Frequency Test

Clocking wizard allows up to two clock inputs and seven clock outputs. It features optional buffers for both input and output clock path. Wizard uses clocking primitive such as Mixed-Mode Clock Manager (MMCM) or Phase-Locked Loop (PLL) to generate a custom HDL code for clock circuits[44]. These clocking circuits can be used to fulfill user's requirement in terms of output clock frequency, phase shift, and duty cycle.

ZYBO incorporates Artix-7 series FPGA which features both MMCM and PLL[50]. PLL provides analog clock management with precise frequency generation. Multiple frequencies of different values can be generated at the same time. However, PLL is unable to provide dynamic and fine phase shifting as featured by DCM (Digital Clock Manager). So, this led to mixed mode of phase shifting (digital) and PLL (analog) to be named as MMCM. MMCM features multiple clock outputs with required phase and frequency in relation to input clock frequency. It also provide options for configurable buffers.

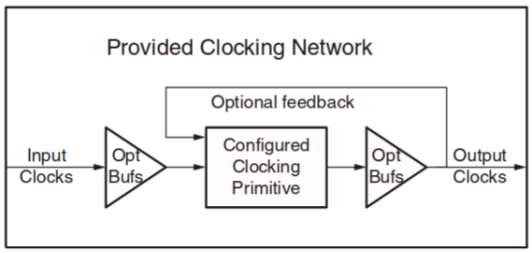
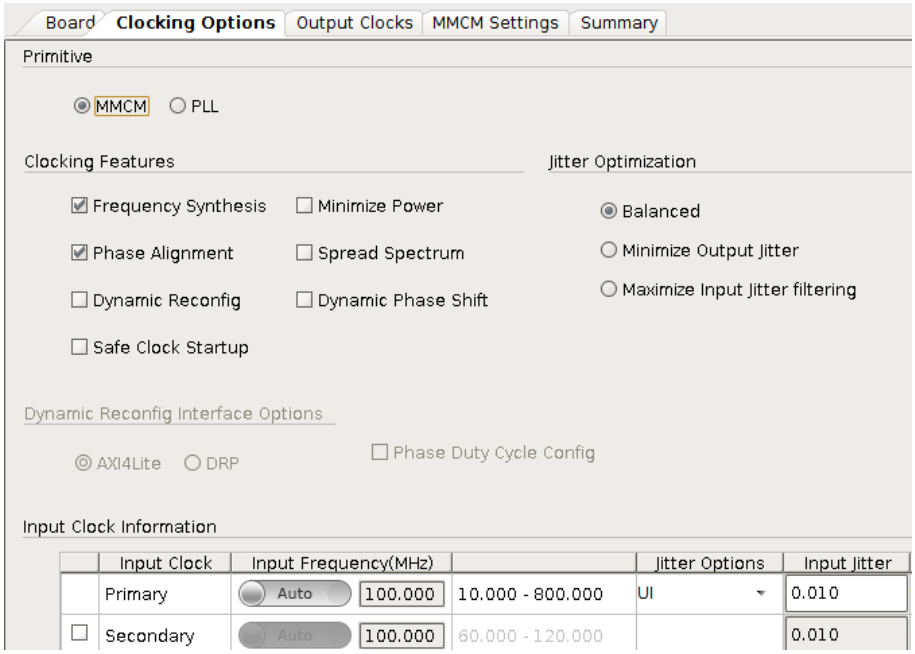


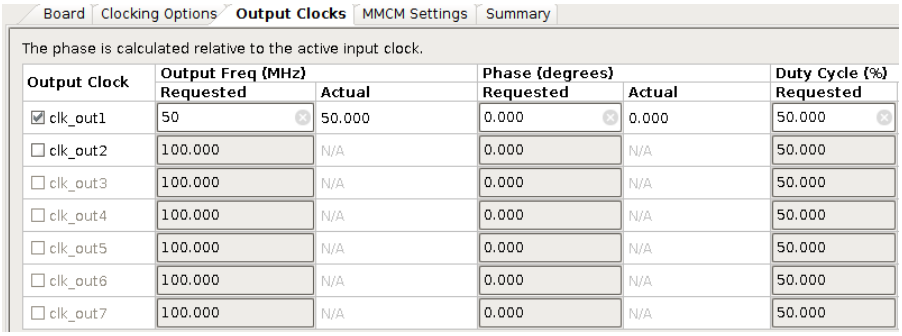
Figure 4.19.: Clocking Wizard Architecture[44]

From Figure 4.18 we see that inputs to the clocking wizard are: *clk_in1* and reset signal. textitclk_in1 is the PL frequency, `FCLK_CLK0`, provided by PS. Clocking wizard use this frequency as a reference for desired frequency.

Outputs of the clocking wizard are: textitclk_out1 and *locked* signal. textitclk_out1 is the desired frequency requested for PL. Phase and duty cycle can also be changed as per requirement to have preferred output. In Figure 4.18 we see that



(a)



(b)

Figure 4.20.: Clocking Wizard (v5.2) Customization

textitclk_out1 is input for Processor System Reset (PSR), Lite IP, clock for GP0, and AXI Interconnect.

Since PSR provides synchronous resets to interconnect and peripherals in PL, therefore, it is needed to be synchronous with clock which is input to PL. GP0 allows PS to perform read/write operation on Lite IP, therefore, port should run at the same clock as Lite IP. Similarly, interconnect is required to provide link between GP0 and Lite IP so same clock frequency is applied to it.

It is possible that textitclk_out1 take some time to stabilize to desired output, therefore, *locked* signal is used for this purpose. When the clock output becomes valid, then *locked* signal is asserted. In the design (Figure 4.18), textitlocked is connected to *dcm_locked* input of PSR. It is connected to *dcm_locked* because it will keep the logic in PL in reset mode until the textitclk_out1 is steady.

Vivado provides customization of clocking wizard as shown in Figure 4.20. As seen in Figure 4.20a, primitive for Figure 4.19 can be selected as MMCM or PLL. Various clock features such as synthesis of frequency, phase alignment for desired clock, dynamic reconfiguration, etc., are highlighted.

Output clocks can be dynamically reconfigured from PS via two interface options: AXI4-Lite and DRP. It also offers settings to minimize jitter in the output clock. From Figure 4.20b, we see that maximum seven clock outputs can be used. Each output clock can be requested for desired frequency, phase and duty cycle. For instance, in Figure 4.20, 100 MHz clock is provided as input to wizard and requested clock output is half of the input clock.

4.2.2. Software Design

For the evaluation of discussed hardware designs, test cases were implemented in C. Each AXI4 interface is tested for read and write operation initiated by PS. Testing is done in terms of time taken to complete the operation for a given input data length. Elapsed time is normalized with respect to corresponding data length. Values of (data length, normalized time) pair are written into a text file for visualization.

For measurement of elapsed time, the GNU (GNU's Not Unix) library declares two data types in *time.h* header file listed as:struct *timepsec* and struct *timeval*[14]. Time measurement for *timeval* are not as precise as *timespec* since it measures time in resolution of 1μs. Time resolution provided by *timepsec* is in nanoseconds. Therefore, in the thesis, struct *timepsec* data type is used for the measurement.

Members of the *timespec* structure are shown in Table 4.2. It offers two struct members: *time_t* and *tv_nsec*. *time_t* stores the elapsed time in seconds whereas *tv_nsec* stores the time in nanoseconds.

Data Type	Identifier	Comment
time_t	tv_sec	seconds
long	tv_nsec	nanoseconds

Table 4.2.: Data Structure of timespec

Linux provides several timing functions as illustrated in Table 4.3. We see that *gettimeofday* and *time* returns wall-clock time. Wall-clock time returns the current time. It does not return the exact CPU time taken for the process. In the thesis, elapsed time is to be measured for read/write process, therefore, wall-clock time is not suitable for this purpose.

clock function provides process time, however, its resolution is precise to maximum 1μs. Similarly, *getrusage* also uses time resolution of 1μs. For better accuracy, *clock_gettime* provides high resolution of 1ns. It provides options for measuring thread or process time.

Time Function	Resolution	Return Value
gettimeofday	1μs	Wall-clock time Timezone
time	1s	Wall-clock time
clock	1μs	Process time
getrusage	1μs	Separate process and system time
clock_gettime	1ns	Wall-clock time Process CPU time Thread-specific CPU time

Table 4.3.: Timing Functions supported by Linux

Apart from the measurement of execution time in relation to data length, effect of PL frequency on execution time is also tested. In the following sections, software design for each AXI4 interface are discussed.

AXI4-Memory Mapped Interfaces

Psuedocode of algorithm for testing performance of AXI4-Full and AXI4-Lite is depicted in algorithm 1.

Inputs to the function are: *slaveAddress*, *slaveLength*, and *clkAvgTime*. *slaveAddress* represents AXI slave IP address which is assigned in Vivado address editor. *slaveLength* corresponds to the number of slave registers in AXI4-Lite slave IP or the size of memory in AXI4-Full slave IP. *clkAvgTime* specifies average time taken solely by *clock_gettime*. Output of the function is a text file that contains values of normalized time in relation to input data length.

MAX_DATA_LENGTH corresponds to maximum number of data allowed to perform operations on AXI slave IP. Operations performed on slave IP are highlighted inside a while loop (line 6 to 12). This loop iterates through each slave register to read or write appropriate value. Index of the slaveAddress are used in modulo of the slaveLength. This is done to keep slaveAddress index within the boundary of registers provided for slave IP. This loop is measured for its execution time to complete the specified number of operations. *varStartTime* and *varStopTime* signifies the time of the start and end of the operations respectively.

arrMeanTime array stores the normalized values of execution time for each *varLength*. As seen in line 17, *varLength* steps in multiples of 10 for each while loop (line 4 to 18). *clkAvgTime* is subtracted from the calculated execution time so that time taken solely by the read/write operation is considered.

For measurement of single AXI slave register/memory, *varLength* is discarded from the equation (line 14). After all operations for maximum data length are performed, results of *arrMeanTime* are written to a text file, *outputData*. Later, this file is used to visualize data through graphs.

Performance of AXI4 memory mapped interfaces is also evaluated in relation to PL frequency. Linux based system features Common Clock Framework (CCF) to

Algorithm 1: Read/Write Performance Test for AXI4 Memory-Mapped Interface

```
1  function axiMemoryMappedTest
   Input  : slaveAddress, slaveLength, and clkAvgTime
   Output: outputData.txt
2  arrLength ← 0
3  while varLength <= MAX_DATA_LENGTH do
4      varStartTime ← Start Timer
5      while index < varLength do
6          if Read Operation then
7              varRead ← slaveAddress(index%slaveLength)
8          else if Write Operation then
9              slaveAddress(index%slaveLength) ← varWrite
10         index ← index + 1
11     end
12     varStopTime ← Stop Timer
13     arrMeanTime(arrLength) ←
         ((varStopTime − varStartTime) − clkAvgTime)/varLength
14     index ← 0
15     arrLength ← arrLength + 1
16     varLength ← varLength ∗ 10
17 end
18 outputData.txt ← stores Normalized Time vs. Data Length values
```

control clocks of the peripherals of SoC design[39]. algorithm 2 illustrates commands used in the system to achieve variable PL frequency in PS. It uses 'devcfg' device driver API to manipulate PL clock frequency (*FCLK0*).

From algorithm 2, *varFreq* denotes the required PL frequency. *system()* is a library function for Linux that provides execution of a specified shell command. Line 2 will create a 'fclk
fclk0' folder with more handles on controlling fclk0[39]. Line 3 enables the use of *FCLK0* device. As seen in line 4, to change PL frequency, set_rate is provided the value of *varFreq*.

Using these system commands, execution time of write operations are measured for respective PL frequency. Range of the frequency used in the test design varies from 0 to 100 MHz.

Algorithm 2: Frequency Test for AXI4 Memory-Mapped Interfaces

```
1 function FrequencyTest
  Input: varFreq
2 system("echo'fclk0' >
   /sys/devices/soc0/amba/f8007000.devcfg/fclk_export")
3 system("echo'1' >
   /sys/devices/soc0/amba/f8007000.devcfg/fclk/fclk0/enable")
4 command ← "echo'varFreq' >
   /sys/devices/soc0/amba/f8007000.devcfg/fclk/fclk0/set_rate")
5 system(command)
```

AXI4-Stream Interface

Testing of AXI4-Stream Interface is done using algorithm highlighted in algorithm 3. Input ot the functions are: *dmareadLite*, *dmaWriteLite*, and *clkAvgTime*. *dmareadLite* and *dmaWriteLite* specifies AXI4-Lite address for read and write DMA respectively (Figure 4.21). These addresses provides PS to send/receive control information from these DMAs. It significance of the output is same as discussed in above section.

Line 3 provides access to physical memory of the system so that operations can be performed on it. While loop (line 4 to 17) implements the time measurement of operation for AXI4-Stream. In order to give an overview of the algorithm, write operation is only illustrated.

Before timing the operation, AXI DMA requires configuration setup such as reset, halt, and details of source and destination address. It features registers to provide such configurations. Registers are displayed in Table 4.4 with their relative addresses.

MM2S Channel	**S2MM Channel**
CONTROL REGISTER (0x00)	CONTROL REGISTER (0x30)
STATUS REGISTER (0x04)	STATUS REGISTER (0x34)
START ADDRESS REGISTER (0x18)	DESTINATION ADDRESS REGISTER (0x48)
LENGTH REGISTER (0x28)	LENGTH REGISTER (0x58)

Table 4.4.: Registers for MM2S and S2MM Channel

After configuration, read and write channels are started such that transfer operation can be performed. S2MM (write) channel depicts flow of data from AXI4-Stream slave IP to DDR memory. MM2S (read) channel allows slave IP to read data from source address in DDR memory.

Operation starts as soon as PS informs about the data transfer length to DMA (line 8 and 10). PS uses LENGTH REGISTER to write the transfer length. After

the completion of operation, DMA generates an interrupt to inform PS of the update. Thereby, synchronization of DMA is needed to perform interrupt handling so that DMA is operational for next set of transfer. PS reads the STATUS REGISTER to read the value of interrupt. In line 9 and 12, *varStartTime* refers to the beginning of the write operation. *varStopTime* registers the time when synchronization of write DMA is completed.

Algorithm 3: Write Performance Test for AXI4 Stream Interface

```
1  function axiStreamTest
   Input  : dmaReadLite, dmaWriteLite, clkAvgTime
   Output: outputData.txt
2  arrLength ← 0
3  Open /dev/mem which represents whole physical memory
4  while varLength <= MAX_DATA_LENGTH do
5      Reset and Halt configuration for both read and write DMA
6      Store source and destination address of DDR memory
7      Start S2MM and MM2S channels
8      Enable Read Transaction (MM2S)
9      varStartTime ← Start Timer
10     Enable Write Transaction (S2MM)
11     S2MM channel synchronization
12     varStopTime ← Stop Timer
13     MM2S channel synchronization
14     arrMeanTime(arrLength) ←
         ((varStopTime − varStartTime) − clkAvgTime)/varLength
15     arrLength ← arrLength + 1
16     varLength ← varLength ∗ 2
17 end
18 outputData.txt ← storesNormalizedTimevs.DataLengthvalues
```

Data length is tested for variation in multiples of 2 and 10. In the current design (algorithm 3), data length steps in multiples of 2. When execution time for every data length is measured, then the result stored in *arrMeanTime* array is written to *outputData.txt*.

For read operation, time measurement is done in two steps. First, elapsed time (t1) is calculated for both read and write operation. In the second step, time taken to perform write operation (t2) is subtracted from the elapsed time ($t1 - t2$).

For different data lengths, values of elapsed times are stored in a file to easily visualize and compare the data for evaluation.

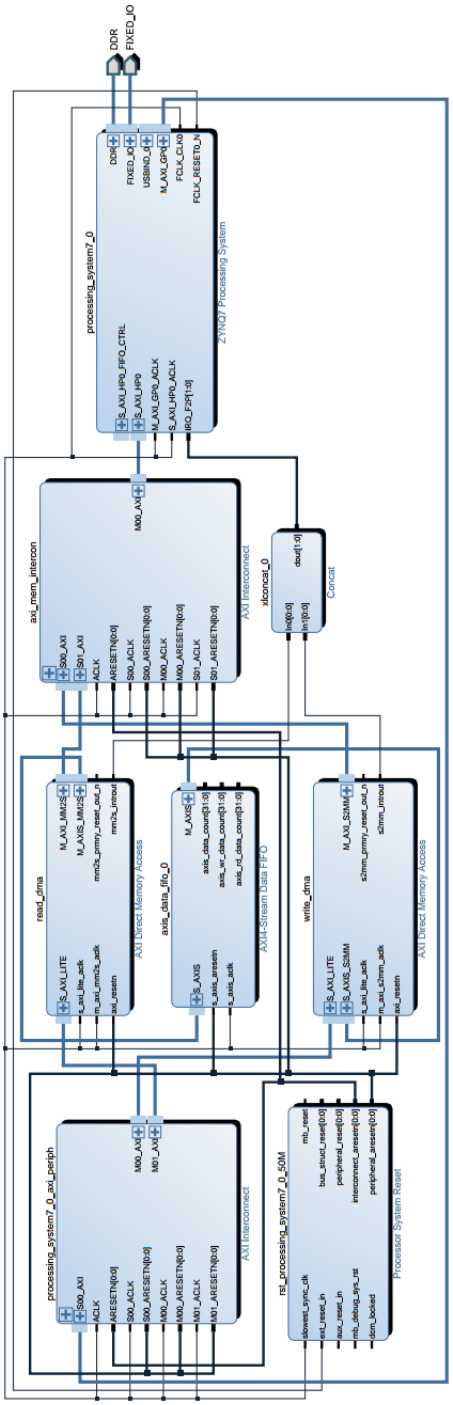


Figure 4.21.: Hardware Design for AXI4-Stream Interface

5. Results and Discussion

As mentioned in the previous chapter, each interface is evaluated for read and write operation. Results of the tests are visualized and later shown in graphs. Some configurations were set to build an executable namely ELF (Executable and Linkable Format) file. This file is transferred over SSH from PC to Linux OS running on PS. Configuration used for generating ELF file and other necessary settings for the results are mentioned in Table 5.1.

Configuration	**Value**
Build type	Release version
Time resolution	1 ns
Transfer data width	32-bit
PL frequency	66 MHz
Optimization level	2 (-O2)

Table 5.1.: Configuration for Results

For the test results, a release build type is used for the generation of ELF file. Results shown to have no significant effect between release and build type. So, release version is preferred to remove any additional debug information from the compiled code.

Xilinx SDK uese GCC (GNU Compiler Collection) for compilation of C code. It provides different optimization levels, from 0 to 3[17], which can have impact on the design perforamnce. In the thesis, optimization level 2 is used. This level, similar to level 1, does not increase the size of executable code. In addition, it supports instruction scheduling. Optimization level 3 is mostly not recommended over 2 since level 3 affects speed-space trade-offs, thereby, probability of increasing the executable code size[17].

In the following subsections, for each AXI4 Interface, results are displayed with and without optimization. For AXI4-Lite IP, 300 slave registers are used for the evaluation. Similarly, for testing, AXI4-Full IP includes 1024 bytes of memory. AXI4-Stream IP deals with the physical memory.

In Figure 5.2 and Figure 5.6, performance of interfaces are calculated in terms of average execution time (Y-axis) with respect to input data length (X-axis).

For memory mapped interfaces, input data length varies from 1 to 10^8 32-bit samples where reading are taken in multiples of 10. For stream interface, input data length varies from 1 to 10^6 data samples. In addition, results for frequency test are displayed in autoreffreqlite. These results are relevant to AXI4-Lite salve IP.

In each displayed graph, first sample value is shown to take significant time compared to other values. This might be due to the initial phase required to setup PS and PL for appropriate read/write transaction.

Performance of the three interfaces for read and write operation are detailed as follows.

5.1. Read Operation

Figure 5.2 represents results for read operation on AXI4 interfaces. As mentioned before, X-axis refers to amount of data used in the read operation. Y-axis denotes average time taken to perform read operation for a given data set.

Figure 5.2a represents read operation on AXI4-Lite slave IP. Time is measured for reading from multiple slave registers. From Figure 5.2a it is evident that after 10,000 data samples, average execution time for not optimized code stabilizes to 210ns. For PL frequency of 66 MHz, it requires 13∼14cc (clock cycles) to read at least 10,000 32-bit data samples over AXI4 Lite interface.

Optimized code in Figure 5.2a seems to have no effect on the time with reference to data length. This is visualized in Figure 5.1 which compares the effect of optimization for a single read operation.

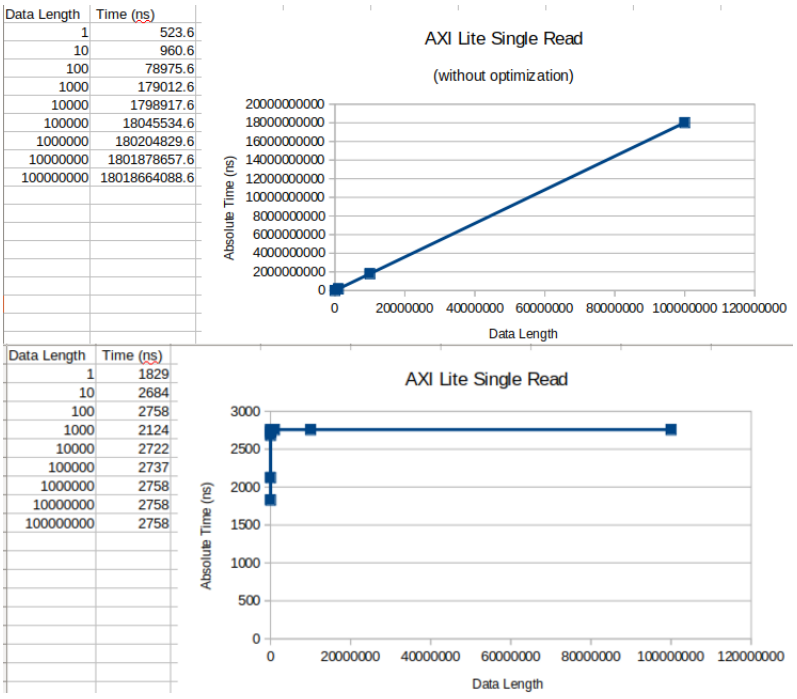
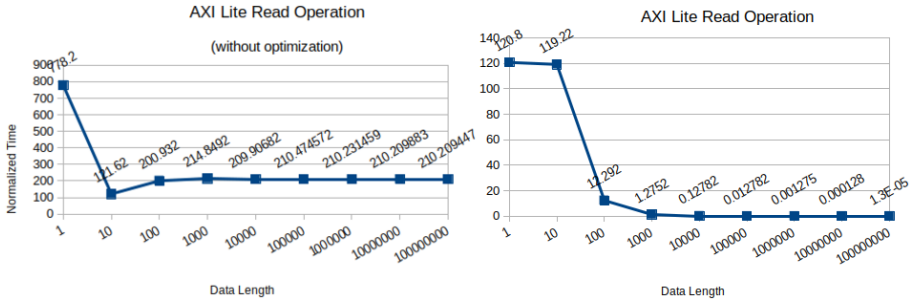
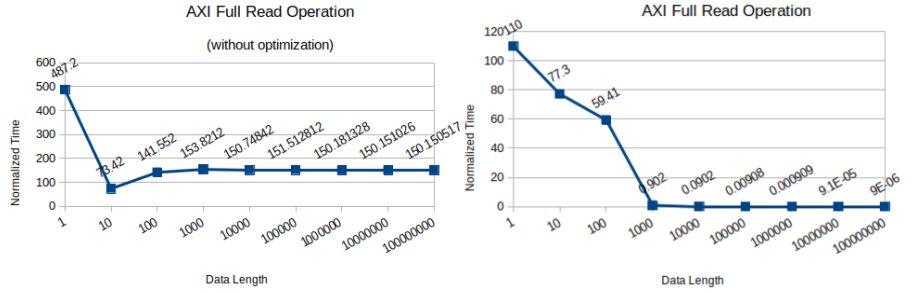


Figure 5.1.: Optimization Effect on the Result

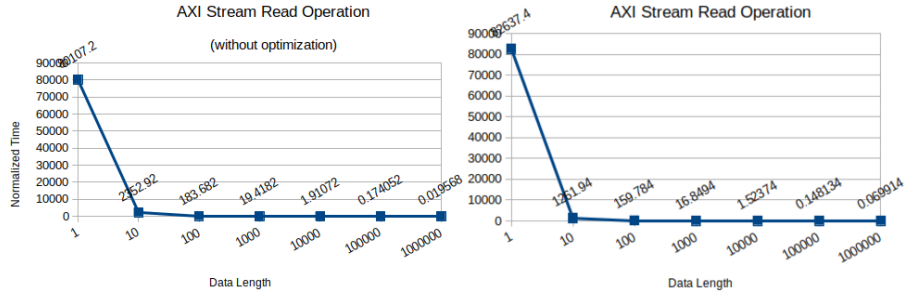
In Figure 5.1, absolute values of execution time are represented. Code without optimization indicates a linear change in execution time as the data increases.



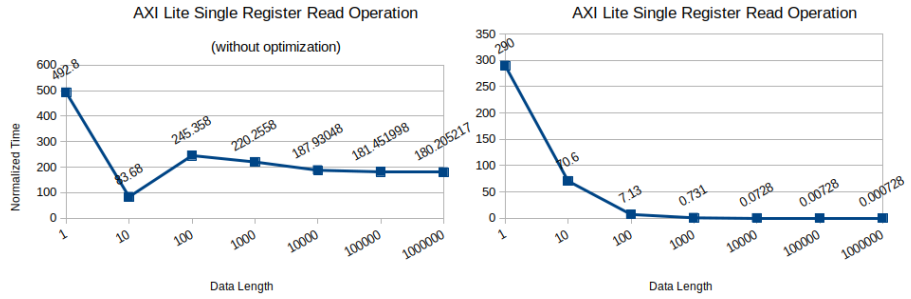
(a)



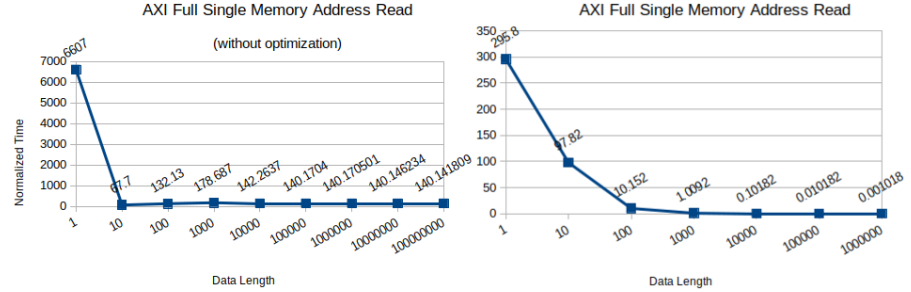
(b)



(c)



(d)



(e)

Figure 5.2.: Results for Read Operations

However, optimized code results in almost no significant change in time with respect to data length. This behavior is identical for all single read/write operations and also for memory mapped multiple read operation.

Reason for such behavior is attributed to the optimization flags used by -O2. To recognize the flags for such response, results were obtained for AXI4-Lite interface with -O1. Outcome of using level 1 featured similar graph pattern as not optimized code in Figure 5.2a. After 10^5 data samples, it resulted a stable value of 2.310175ns. This outcome reduced the sample space for possible optimization flags solely related to -O2. According to [17], -O2 utilizes all optimization flags supported by -O1 and, additionally, incorporates flags for instruction scheduling. For more details, these additional flags are listed in Appendix A.

In Figure 5.2b, time measurements are registered for reading from multiple addresses of 1024 bytes of slave memory. We see that AXI4 Full interface without optimization results in stable value of 150ns (9∼10cc). Behavior for optimized code is similar as discussed before, time is shown to be independent of the data length. Therefore, the graph declines as the data length increases.

For AXI4-Stream (Figure 5.2c), normalized time significantly decreases as the input data length increases. To read data off the stream, optimized code performs better in comparison to code without optimization. After 1000 data samples, both codes shown no change in the time value. This is because of the AXI4-Stream Data FIFO buffer length set to 4096 values. In next section, result for AXI4-Stream are shown for write operation so that it does not exceed the FIFO buffer length.

Single read from memory/register is also depicted in Figure 5.2d and Figure 5.2e. In Figure 5.2d, AXI4-Lite requires average time of 180ns (11∼12cc) while the optimized code shows constant change in time with reference to data samples. For code without optimization (Figure 5.2e), AXI4-Full results in 140ns (9∼10cc).

From the results seen in Figure 5.2, we see that, for both single and multiple read, AXI4-Full performs faster (by 2cc) in comparison to AXI4-Lite. However, optimized code for single read operation provides faster execution time for AXI4-Lite than AXI4-Full.

From Figure 5.2b, for multiple and single read operation, AXI4-Full performs faster compared to AXI4-Lite (for not optimized code). Full is 2cc faster than Lite for single read operation whereas 4cc faster for multiple read operation.

For not optimized code, in Figure 5.2c, we see that for 1000 data samples stream interface is the fastest compared to memory mapped interface. Stream results in 19.4182 ns whereas Full and Lite takes 150 ns and 210 ns respectively. The amount of difference in execution time is significant.

5.2. Write Operation

Results for write operations are depicted in Figure 5.6. From Figure 5.6a, we see graph for AXI lite write operation performed on multiple slave registers. Optimized code has a better execution of 180ns (11∼12cc) compared to not optimized code

which stabilizes at 240ns ((15∼16cc). Code without optimization is 33.33% slower than the optimized code.

In Figure 5.6b, write operations are executed on multiple addresses of slave memory. It highlights same behavior as above in the case that optimization makes the execution time 33% faster than its counterpart. Optimized code runs at 170ns (11∼12cc) whereas the counterpart stabilizes at 130ns (8∼9cc).

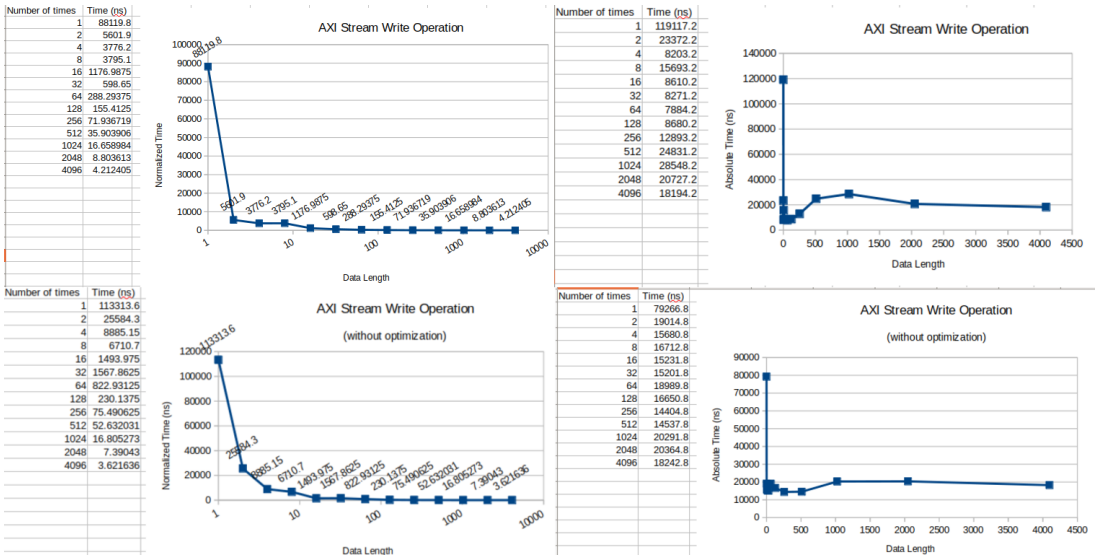


Figure 5.3.: Comparison of Normalized and Absolute Time for AXI4 Stream

In Figure 5.6c, in both the cases, the time values are significantly low as the data increases. That is, it is too fast for large data samples which is not the normal behavior. As mentioned in section 5.1, this can be due to FIFO depth of size 4096. Such that after 1K samples, mean time remains the same as data length increases in multiples of 10 from 1K. So, large values will not be stored in FIFO buffer, thereby, consisting of the same amount of data.

So a maximum data length of 4K is tested for data step in multiples of 2. Results for this test are shown in Figure 5.3. 'Number of Times' in Figure 5.3 refers to data length. Testing is implemented for AXI4-Stream multiple write operation. Results for both configuration of optimization are displayed. In addition to normalized time, absolute time is also calculated and displayed. This was done to check if the values are constant after certain data length.

From outlook, both optimized and not optimized code show similar behavior. We see that, for optimized code, absolute time reduces after 1K and then becomes constant. For the not optimized, time after 1K is constant and then a slight reduction in time. So we see that absolute time should increase after 1K similar to the linear behavior of absolute values for single read operation (Figure 5.1).

From Figure 5.6d we see that for write operation to a single slave register, AXI4 Lite provides best execution time of 210ns (13∼14cc). Optimized code shows similar behavior as discussed before for read operation. From Figure 5.4 we see that, except

for two data samples, absolute time is almost constant. AXI4-Lite shows same characteristics as Figure 5.2d. Optimized code uses similar amount of execution time for both single slave register for read/write operation. However, without optimization, write operation takes 1 extra clock cycle for read operation.

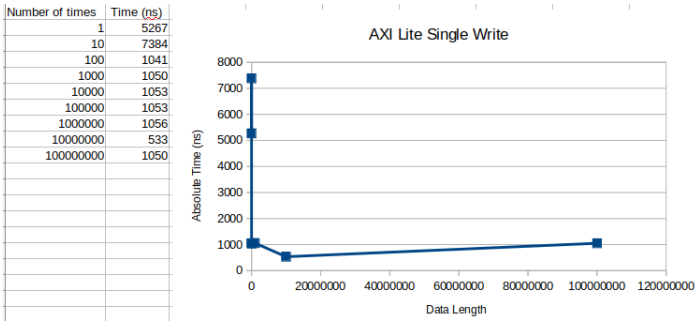


Figure 5.4.: Absolute Execution Time for AXI4-Lite Write Operation

Same behavior for optimized code in Figure 5.6e is depicted. To write to a single slave address, code without optimization provides stable time of 160ns (10∼11cc).

In overall, we see that, for both cases of optimization, AXI4-Full performs faster for both multiple address and single address write operation. For multiple write operation, Full IP is faster by 4cc when code is not optimized. And when optimized, it is faster by 3cc. Both AXI4-Lite (Figure 5.6a) and AXI4-Full (Figure 5.6b) provides optimized code which is 33.33% faster than the counterpart.

In Figure 5.6c, optimized code runs reasonable slower than the not optimized code. For write operation, optimized code takes slightly more time than not optimized code. This is not the case for read operation. Also it can be seen that writing is faster than reading the stream, for both with and without optimization.

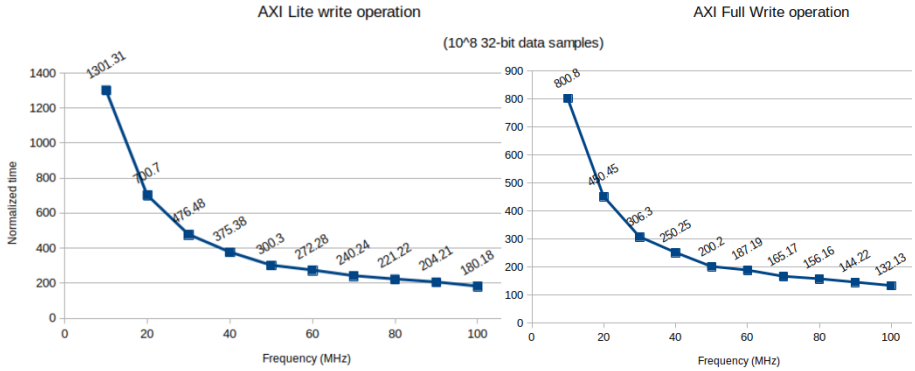
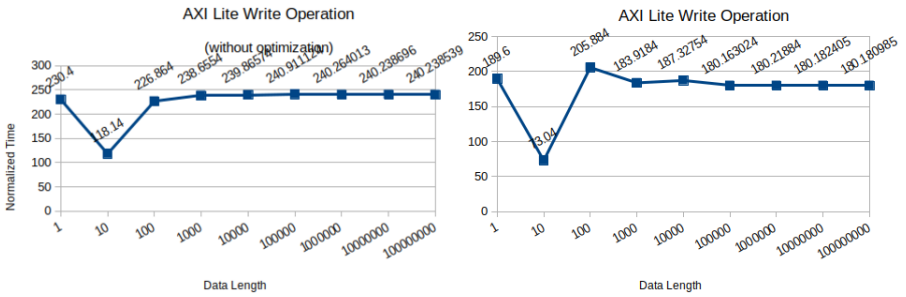
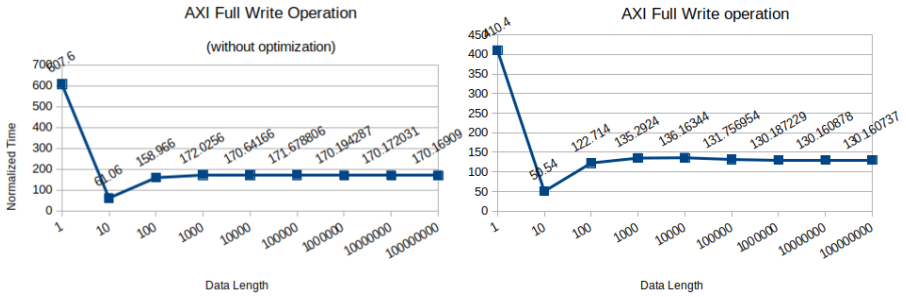


Figure 5.5.: Performance based on PL Frequency

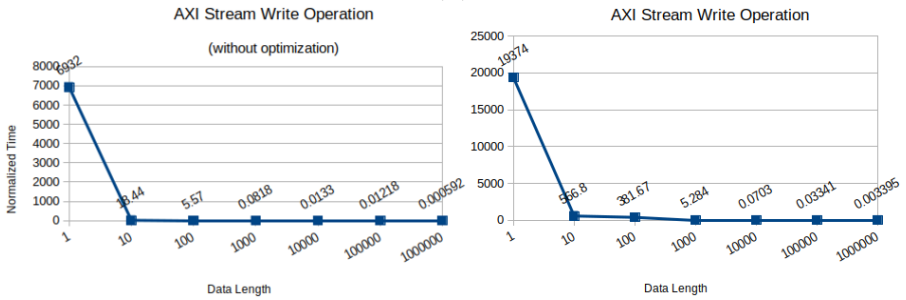
Time taken by stream interface for 1000 data samples (Figure 5.3) is around 16ns. This is significantly faster than optimized code for memory mapped interfaces (Figure 5.6a and Figure 5.6b). It seems AXI4-Stream interface has advantage for burst operations owing to its protocol framework without the use of addresses and few control signals.



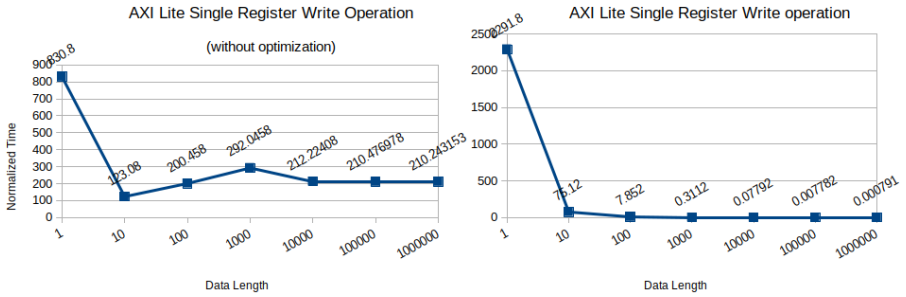
(a)



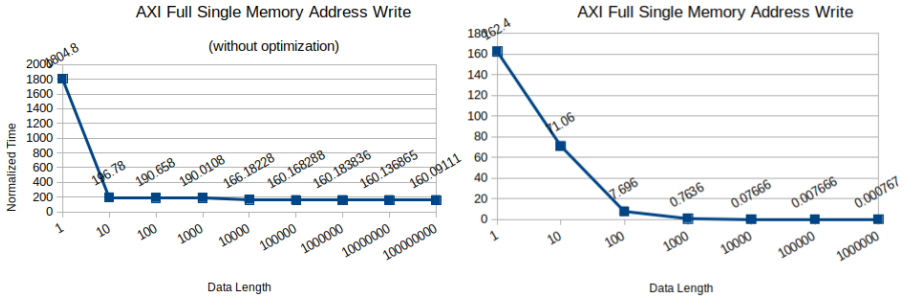
(b)



(c)



(d)



(e)

Figure 5.6.: Results for Write Operation

In addition, time for multiple write operations is evaluated with reference to change in frequency. Behavior of both AXI4-Lite and AXI4-Full interface are tested for different PL input frequency. Evaluation is done with 10^8 data samples.

Results are highlighted in Figure 5.5. It is seen that execution time shows inverse relationship with PL frequency. As the frequency increases, mean time to write data to slave register/memory decreases.

5.3. Comparison of Results

Results of interfaces discussed in the previous section are summarized here in Table 5.2.

	AXI4-LITE			AXI4-Full			AXI4-Stream		
	Max. Data Length	Optimized	Not Optimized	Max. Data Length	Optimized	Not Optimized	Max. Data Length	Optimized	Not Optimized
Read Operation	10^8	Independent of input length (slight change in mean time with input length)	210ns (13∼14cc)	10^8	Independent of input length (slight change in mean time with input length)	150ns (9∼10cc)	10^6	Completion Time inversely proportional to input length (17ns for 1000 data samples)	Completion Time inversely proportional to input length (19ns for 1000 data samples)
Write Operation	10^8	180ns (11 - 12cc)	240ns (15∼16cc)	10^8	130ns (8∼9cc)	170ns (11∼12cc)	10^6	5.284ns for 1000 data samples	2.458 for 1000 data samples
Single Read Operation	10^6	Independent of input length (slight change in mean time with input length)	180ns (11∼12cc)	10^6	Independent of input length, slightly slower than AXI4-Lite	140ns (9∼10cc)	-	-	-
Single Write Operation	10^6	Independent of input length (slight change in mean time with input length)	210ns (13∼14cc)	10^6	Independent of input length, slightly slower than AXI4-Lite	160ns (10∼11cc)	-	-	-

Table 5.2.: Comparison of AXI4-Interface Results

In the Table 5.2, 'Read Operation' and 'Write Operation' denotes multiple read-/write operation on slave register/memory. 'Single Read Operation' and 'Single Write operation' signifies read/write to a single slave register/memory. Performance of these operation are evaluated with reference to the amount of data involved in the operation. Best execution time for these operations are highlighted for comparison.

We see that optimized code for all interfaces, except for multiple write operation, results in only slight change in time with respect to data length. So, from given results, code without optimization is compared to evaluate the fitting interface for the thesis.

For read and write operation, it is seen that AXI4-Full performs better in all cases as compared to AXI4-Lite. It is faster by 4cc for read and write operation. For single read and write operation, it is efficient by 2cc and 3cc respectively.

Results for AXI4-Stream are displayed for 1000 samples. As mentioned in the previous section, Stream offers the best execution time compared to memory mapped interfaces. For instance, for 1000 samples, it offers time for not optimized code of 19ns wheres AXI4-Full offers 150ns. We see that the difference between time execution is significant.

However, for write operation, code without optimization gives better performance for 1000 data samples. Also it is seen that execution time is unrealistically fast for large number of data. Therefore, stream test is done for 1024 data samples with data steps in multiples of 2. Result is displayed in Figure 5.3. From the graphs of optimized and without optimized, it is seen that normalized time comes out to be similar. That is, there is hardly an impact of optimization for write operation.

As previously mentioned, absolute time after 1k samples becomes constant or decrease. Ideally, the time should increase. Reason for this behavior might be related to AXI DataMover IP which is already included in AXI DMA. DataMover features an internal FIFO to store the data with the same frequency as DDR memory. This FIFO, by default, uses the same clock as the memory mapped data to function synchronously. It can also be used as asynchronous FIFO to be clocked differently than memory mapped interface[47].

So it might be possible that DDR access this internal buffer at much faster clock-domain for DDR, which results in the decrease of absolute time. This is still a hypothesis which is yet to be tested.

6. Conclusion and Further Work

Conclusion

Three different AXI4-Interfaces have been evaluated for its performance in terms of processing time and input data length. To perform the evaluation, an FPGA implementation for the hardware was provided and a machine learning interface.

A SoC hardware board, ZYBO, was used to do the time measurements. Hardware for the time measurement is designed in Vivado and tested on ZYNQ. Testing is done by test cases implemented in Xilinx SDK. Hence, three hardware designs were implemented as listed: AXI4-Lite, AXI4-Full, and AXI4-Stream. These designs were tested for read and write operation.

Multiple read/write operations were performed on: 300 slave registers for Lite, 1024 bytes of memory for Full, and 4k Data FIFO for Stream. Single read/write operations were performed on memory-mapped interfaces. Dependence of execution time on PL frequency is also tested for memory-mapped interfaces. To highlight such dependence, testing was done for multiple write operations.

From the discussed results, it can be said that, for memory-mapped interfaces, AXI4-Full is a fitting interface due to its efficient execution time. It performed better than AXI4-Lite for both read and write operation. When AXI4-Full is compared to AXI4-Stream, then Stream appears to be superior because of significant faster execution time. It shows to be a fitting choice for the provided hardware given that feature data from PL to PS is to be transferred in burst.

However, reason for the reduction of execution time for AXI4-Stream is still unknown. Possible hypothesis is AXI DataMover which acts as an internal buffer for AXI DMA, might buffer transfer data as soon as termination flag is set. Also, by default, this buffer runs synchronously with DDR faster clock-domain. This might lead to decrease in execution time.

In conclusion, based on the needs of the MERGE project, AXI4-Stream turns out to be opt-choice for continuous burst data transfer. Also because of the usage of High-Performance data bus, it shows better results than memory-mapped interfaces. Memory-mapped interface was provided by General Purpose port which features a low performance bus. Due to this reason, an AXI4-Full IP (Figure B.1) was designed to communicate with High-Performance bus and incorporate burst mode. However, this IP is not tested in the thesis but might be slower compared to DMA transfer speed.

Therefore, for real-time scenario where continuous unlimited burst is a need, AXI4-Stream is the viable option. When it comes to memory-mapped interface,

AXI4-Full offers better performance compared to AXI4-Lite. It supports burst configuration but with a limit of 256 beats per burst.

Further Work

As mentioned in the conclusion, performance test for an AXI4-Full design in appendix B is to be undertaken. Reason for this is the use of High-Performance Port (HP0) which provides FIFO buffer support and higher data width compared to GP0.

In the result section, frequency tests for AXI4-Full and AXI4-Lite are discussed. Figure 6.1 highlights performance of AXI4-Stream with respect to frequency. Time calculation is done for 100 data samples read from DDR memory.

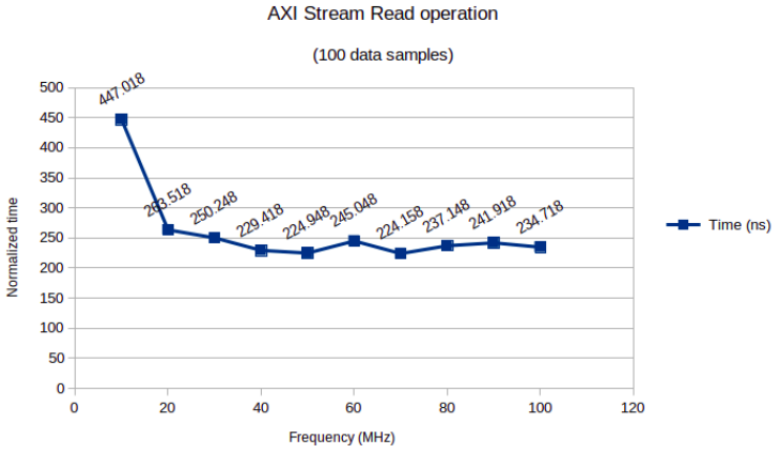


Figure 6.1.: AXI4-Stream Performance Relative to PL Frequency

The outcome is not as expected. With increase in frequency, there is only a slight change in the execution time. Normally, execution time should decrease as the number of clock cycles, provided for the same task, increases. Therefore, reason for this behavior are still needed to be found.

Also, as mentioned before in conclusion, the hypothesis proposed for the decrease of execution time for AXI4-Stream has to be tested.

In the thesis, AXI4-Full is not tested for its different data width and burst configuration. However, it is discussed in Figure 4.12 in section 4.2. For AXI4-Full, 64-bit and 128-bit data widths were tested. However, it resulted in increase of average execution time compared to time needed for 32-bit data width. As seen from Figure 6.2, for 64-/128-bit data width it takes 160ns to do multiple write operation for the optimized code. Whereas, in result section, Figure 5.6b provides 130ns of execution time.

Possible reasoning for such result is the inclusion of AXI Data Width Converter IP in AXI Interconnect which provides connection between GP0 and AXI4-Full slave IP(autoreffig:fulldata). Since GP0 supports 32-bit, therefore, because of the conversion process there is an increased time in processing higher data width.

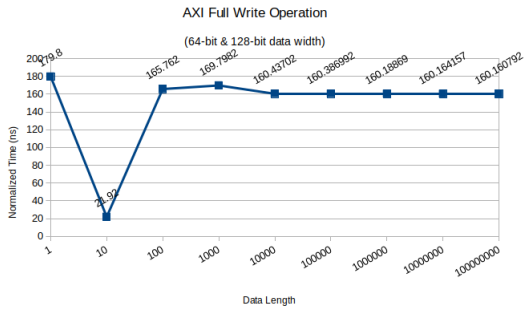


Figure 6.2.: AXI4-Full Performance for 64-/128-bit Data Width

One possible solution would be to use AXI CDMA to perform different data width and burst configuration. A proposed hardware design is provided in appendix D. It provides selection for data width from 32 to 1024 bits and burst size from 2 to 256.

Bibliography

[1] The Coreconnect Bus Architecture, http://www.scarpaz.com/2100-papers/SystemOnChip/ibm_core_connect_whitepaper.pdf

[2] Universal Serial Bus 3.2 Specification, 1st edn. (September 2017), https://www.usb.org/document-library/usb-32-specification-released-september-22-2017-and-ecns

[3] (January 2018), https://projects-raspberry.com/introduction-i%C2%B2c-spi-protocols/

[4] Accellera: Open Core Protocol Specification (2013), http://www.accellera.org/downloads/standards/ocp

[5] Altera: Avalon Bus Specification (July 2002), http://coen.boisestate.edu/clarenceplanting/files/2011/09/avalon_bus_spec.pdf

[6] ARM: AMBA Specification 2.0 (May 1999), http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html

[7] ARM: AMBA AXI and ACE Protocol Specification (December 2017), https://developer.arm.com/docs/ihi0022/latest/amba-axi-and-ace-protocol-specification-axi3-axi4-axi5-ace-and-ace5

[8] Bosch: CAN Specification 2.0 (1991), http://esd.cs.ucr.edu/webres/can20.pdf

[9] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips: Universal Serial Bus Specification, 2nd edn. (april 2000), http://sdphca.ucsd.edu/Lab_Equip_Manuals/usb_20.pdf

[10] Cook, J., Freudenberg, J.: Controller Area Network (CAN) (2008), https://www.ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Lectures/Embedded-Control-Systems/OtherNotes/CAN_notes.pdf

[11] Corporation, N.I.: Controller Area Network (CAN) tutorial, http://download.ni.com/pub/devzone/tut/can_tutorial.pdf

[12] Corrigan, S.: Introduction to the Controller Area Network (CAN) (May 2016), http://www.ti.com/lit/an/sloa101b/sloa101b.pdf

[13] Digilent: ZYBO FPGA Board Reference Manual (February 2017), https://reference.digilentinc.com/_media/reference/programmable-logic/zybo/zybo_rm.pdf

[14] Free Software Foundation, I.: http://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html

[15] Free Software Foundation, I.: Online, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[16] Gaur, M.S., Laxmi, V., Zwolinski, M., Kumar, M., Gupta, N., Ashish: Network-on-Chip: Current issues and challenges. In: Proc. 19th Int. Symp. VLSI Design and Test. pp. 1–3 (June 2015)

[17] Gough, B.J., Stallman, R.M.: An Introduction to GCC: For the GNU Compilers GCC and G++. Network Theory Ltd., 2nd edn. (March 2004)

[18] IBM: 32-bit Processor Local Bus: Architecure Specification (May 2001), https://ptolemy.berkeley.edu/projects/embedded/mescal/forum/7/coreconnect_32bit.pdf

[19] IBM: On-Chip Peripheral Bus Architecture Specifications (2001), http://www.cs.columbia.edu/~sedwards/classes/2005/emsys-summer/opb_ibm_spec.pdf

[20] Instruments, T.: Interface Circuits for TIA/EIA-232-F (September 2002), http://www.ti.com/lit/an/slla037a/slla037a.pdf

[21] Instruments, T.: Interface Circuits for TIA/EIA-485 (RS-485). Tech. Rep. SLLA036D (August 2008), http://www.ti.com/lit/an/slla036d/slla036d.pdf

[22] Instruments, T.: RS-422 and RS-485 Standards Overview and System Configurations (May 2010), http://www.ti.com/lit/an/slla070d/slla070d.pdf

[23] Instruments, T.: AN-1031 TIA/EIA-422-B Overview (April 2013), http://www.ti.com/lit/an/snla044b/snla044b.pdf

[24] Intel: Avalon Interface Specifications (September 2018), https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[25] Irazabal, J.M., Blozis, S.: AN10216-01 I2C Manual. Philips Semiconductors (March 2003), https://www.nxp.com/docs/en/application-note/AN10216.pdf

[26] L-com: What is USB?, https://www.l-com.com/multimedia/tips/tip_what_is_usb.pdf

[27] Miti, M., Stojcev, M.: An Overview of On-Chip Buses. ResearchGate (January 2006), https://www.researchgate.net/publication/237587431_An_Overview_of_On-Chip_Buses

[28] Murphy, R.: USB 101: An Introduction to Universal Serial Bus 2.0. Tech. rep., Cypress (April 2017), http://www.cypress.com/file/134171/download

[29] NXP Semiconductors: I2C Bus Specification and User Manual, 6th edn. (April 2014), https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[30] OCP: Open Core Protocol, https://www.accellera.org/images/community/ocp/datasheets/OCP_30_Datasheet.pdf

[31] Omega: The RS-232 Standard, https://www.omega.de/techref/pdf/RS-232.pdf

[32] OpenCores: Wishbone B4: WISHBONE System-on-Chip (SoC)Interconnection Architecture for Portable IP Cores (2010), https://cdn.opencores.org/downloads/wbspec_b4.pdf

[33] Pasricha, S., Dutt, N.: On-Chip Communication Architectures: System on Chip Interconnect. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)

[34] Siemens: Controller Area Network (October 1998), http://ecee.colorado.edu/~mcclurel/CANPRES.pdf

[35] Sinha, R., Roop, P., Basu, S.: The AMBA SOC Platform. In: Correct-by-Construction Approaches for SoC Design. Springer (January 2014), http://dx.doi.org/10.1007/978-1-4614-7864-5_2

[36] Stevens, A.: Introduction to AMBA 4 ACE and big.LITTLE Processing Technology (July 2013), https://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf

[37] Ullmann, F., Hardt, W., Zhmud, V.: Machine learning algorithms for impact localization on formed piezo metal composites. In: 2017 International Siberian Conference on Control and Communications (SIBCON). pp. 1–5 (June 2017)

[38] Ullmann, F., Hardt, W.: Towards Impact Detection and Localization on a Piezo Metal Composite (July 2016)

[39] Wiki, X.: Online, https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841795/Controlling+FCLKs+in+Linux

[40] Xilinx: Software Development Kit. Online, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/SDK_Doc/index.html

[41] Xilinx: AMBA 4 AXI4-Stream Protocol Specification, 1st edn. (March 2010), http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html

[42] Xilinx: AXI Reference Guide, 13th edn. (March 2011), https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[43] Xilinx: Zynq Architecture (2012), http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf

[44] Xilinx: Clocking Wizard v5.2 (Novemeber 2015), https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_2/pg065-clk-wiz.pdf

[45] Xilinx: AXI4-Stream FIFO v4.1 (April 2016), https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf

[46] Xilinx: Vivado Design Suite User Guide Implementation (April 2016), https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug904-vivado-implementation.pdf

[47] Xilinx: (April 2017), https://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf

[48] Xilinx: AXI Interconnect v2.1 (December 2017), https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[49] Xilinx: 7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter (July 2018), https://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf

[50] Xilinx: 7 series fpgas data sheet: Overview (February 2018), https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[51] Xilinx: ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC (June 2018), https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

[52] Xilinx: Zynq-7000 SoC Technical Reference Manual (July 2018), https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

A. Optimization level 2 flags

Below is the list of optimization flags used by -O2 and are not supported in -O1[15]

-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-pre
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp

B. AXI4-Full Test Design

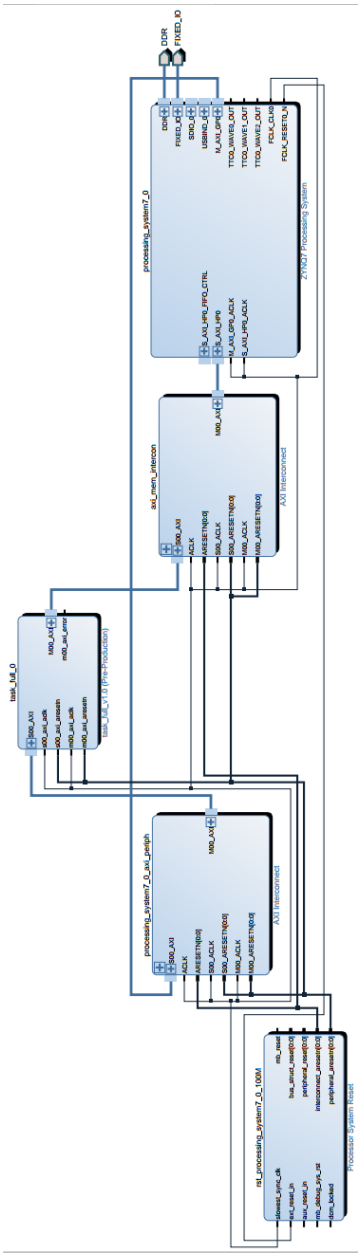


Figure B.1.: AXI4-Full High-Performance Based Design

C. AXI4-Stream Clocking Wizard Design

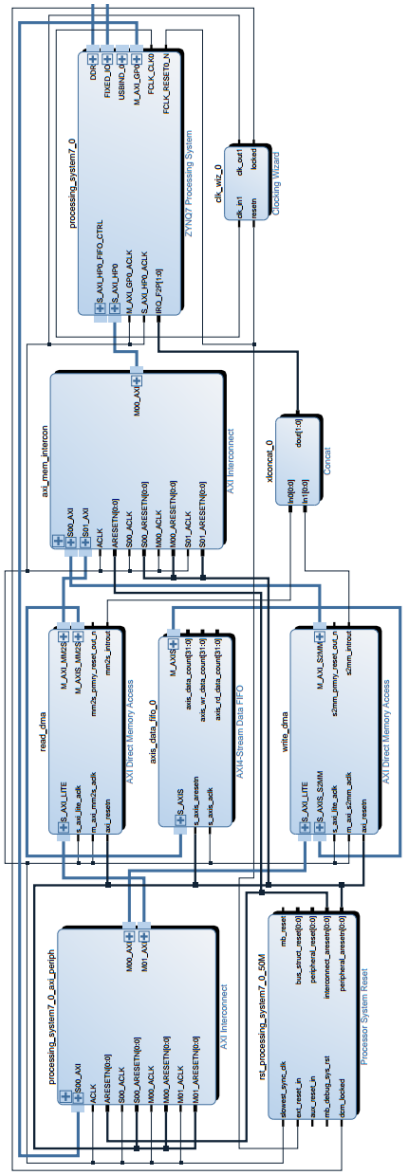


Figure C.1.: Clocking Wizard design for AXI4-Stream

D. AXI4-Full CDMA Design

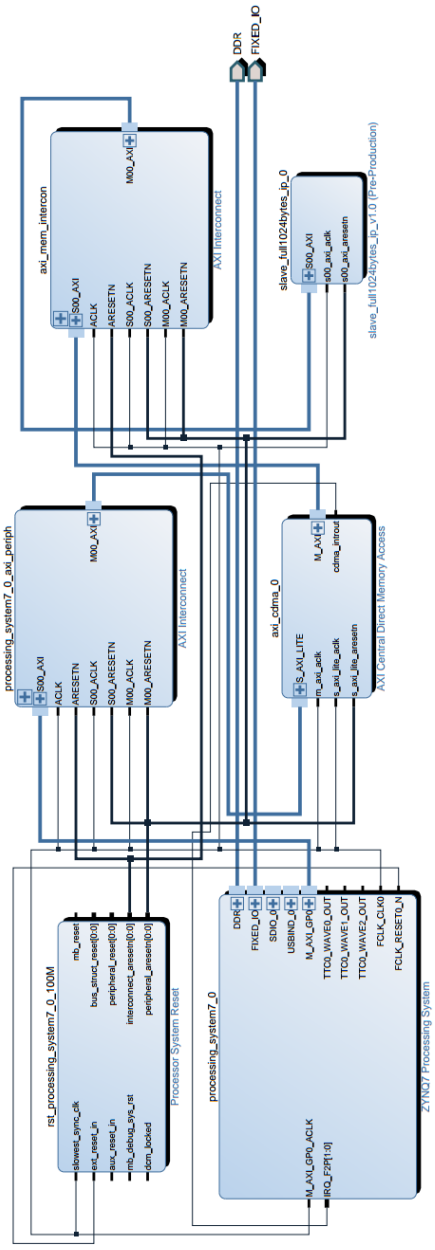


Figure D.1.: AXI CDMA Based Hardware Design