UPPSALA
UNIVERSITET

Data Acquisition System for the ZynqBoard

Author: *Sina Borrami*

Supervisor: *Pawel Marciniewski*

2019-11-30

Abstract:

Zynq is a new System on chip (SOC) family of Xilinx, which features an FPGA with a programmable logic (PL) and processing system (PS). The cores are Cortex-A9 with ARMv7-A architecture running in the range of 600-1000 MHz. In this project, the ZynqBoard which utilizes the Zynq SOC is used and the possible implementations of a data acquisition system in order to achieve the best throughput of data over the Gigabit Ethernet port is studied.

In this study, the logic for connecting the ZynqBoard to PANDA experiment's ADC boards for retrieving data and syncing them, is implemented and various methods of data movement from PL to PS in the Zynq Architecture are explored.

Finally, on the recipient host which is a Linux platform, a simple server application for receiving and storing stream of data is developed. The firmware is implemented with LWIP library in the FreeRTOS platform and VHDL and the data with the 65MB/s throughput were transferred and stored.

Contents

1 Introduction

FPGAs are widely used in experimental setups for high energy particle physics. The FPGAs provide the best platform to physicists to have a reconfigurable chip. This reconfigurability reduces a test time and gives them the capability to modify the firmware based on each experiment. In addition, almost all medium to advance FPGAs offer high speed transceivers which can have a speed up to 400Gbps. This feature gives the scientist the capability to transmit the data with high speed commutations and also provide them the additional feature to sync two point-to-point connection FPGAs to the fraction of Nano-second. This sync mechanism provides the physicists with affordable and feasible distributed acquisition system. Generally, data transmissions are done via costume protocol to build the uplink and sync mechanism. To monitor and analyze the data in human readable format, there is a need for gateways. This gateway should convert the data from the costume protocol to computer standard data bus.

Generally, in such experimental setups, rack mount power crates such as VME, VXS and VXI with sophisticated boards for controlling and acquisition of distributed boards in the experimental setups are used. There is a lot of drawback to use cates. The main issues are, the boards should be crate compatible, the crates backbone bus is limited, working with crates controllers are cumbersome and its drivers are not supported by the modern operating systems.

To overcome crates issues, ZynqBoard as an alternative design at Uppsala University was developed. It is a Zynq device with four SFP channel inputs and one Gigabit Ethernet which is a suitable board for acquisition purposes.

In this project to make the use of the ZynqBoard, a firmware based on the specifications of the PANDA experiment's ADC board is developed.

2 Project Description

In this project, it is intended to develop a firmware for the ZynqBoard gateway in VHDL and C language. The Zynq-Board should receive the data from its four fiber transceivers in a fair deterministic manner[1] and transfer the collected data through the data movers from PL to PS in the Zynq SOC FPGA based on the capabilities of the mounted chip and finally send the collected data via TCP protocol to the server. As shown in the Figure 1, the Analog to Digital Convertor (ADC) boards which are developed at the Uppsala University as an acquisition system for the PANDA experiment [1], are connected to ZynqBoard with high-speed serial transceivers via Multi-Mode (MM) Small Form-factor Pluggable transceivers (SFP). To store the read data, the ZynqBoard is connected to Linux host PC via Ethernet cable.

The server application, after running will bond on the desired TCP port on the host and Zynq-Board will connect to the server via Gigabit Ethernet cable. The Zynq-Board after acquiring the data from the ADC boards will transmit the data to the host and all payloads will be stored in the binary raw format on the storage disk.

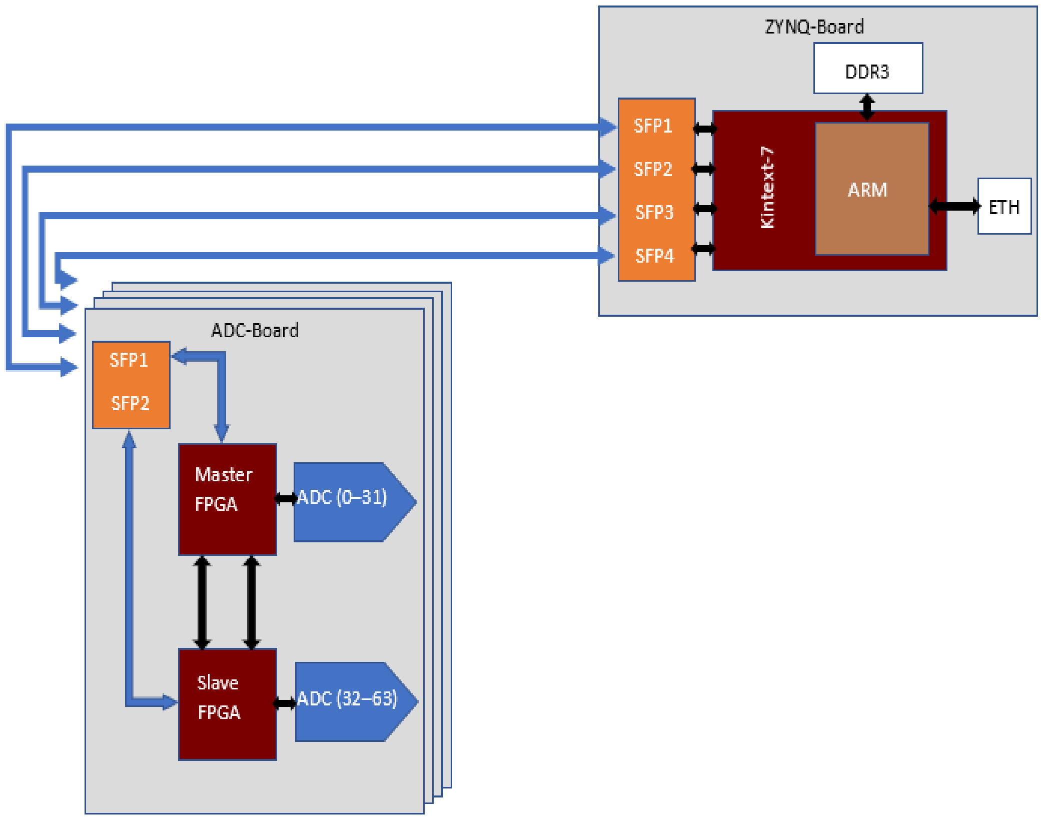


Figure 1-Overview

[1] Each transmitter has the same opportunity as the other ones to transmit data

2.1 Tools and Software

For designing, implementing and testing the firmware for ZynqBoard the following tools has been used:

a. Xilinx Vivado HLS
b. Xilinx Vivado
c. Xilinx SDK
d. Xilinx Chip scope

2.2 Hardware

For exploring to find the best method of moving data from PS to PL and ethernet socket programming, the ZedBoard evaluation kit as shown in Figure 2 was the main platform. Later the ZynqBoard and the ADC board were used to complete the design.

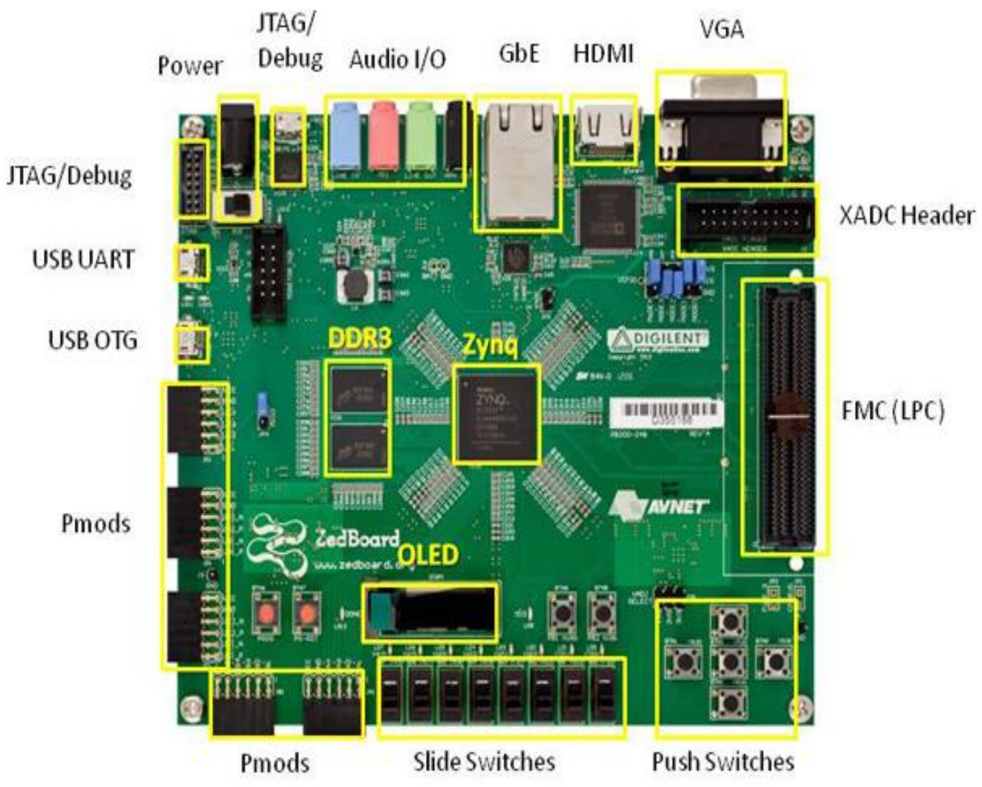


Figure 2-ZedBoard

As it can be seen in Figure 3, the ZynqBoard has a four SFP cage that can simultaneously connect to four ADC boards to collect and process data. Moreover, at the PS side it enhances 512MB DDR3 RAM and 1 Gigabit Ethernet to connect to the network via TCP protocol.

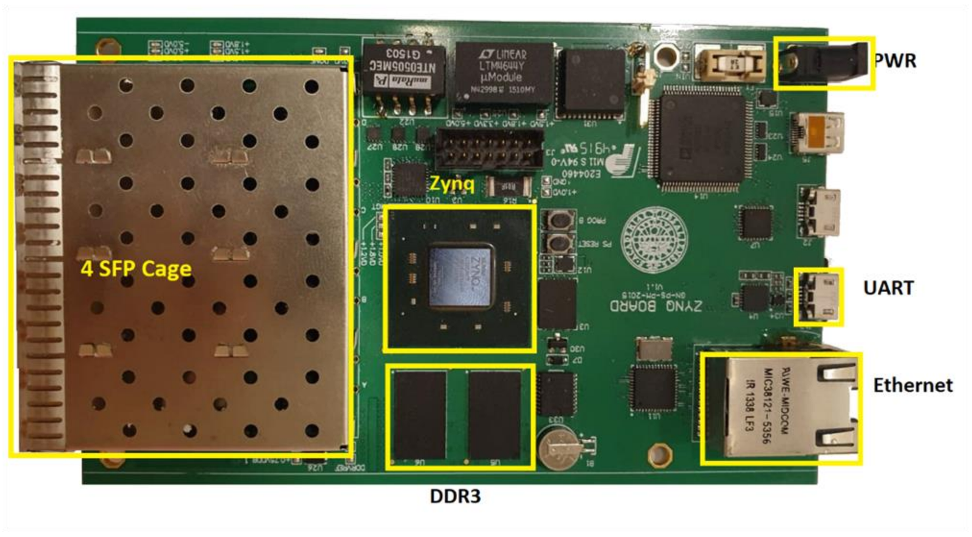


Figure 3-ZynqBoard

As it can be seen in Figure 4, ADC board has 64 channels of differencial 14-bit resolution ADCs with 80 Mbps Sample Rate. The channels are distributed equally between two available FPGAs to maintain redundancy. FPGAs are connected internally with two high-speed serial tranceivers and each FPGA is individually connected to each SFP cage. With having distributed design, it will provide more tolerant system and if any of FPGAs fails, half of the channels are secured.

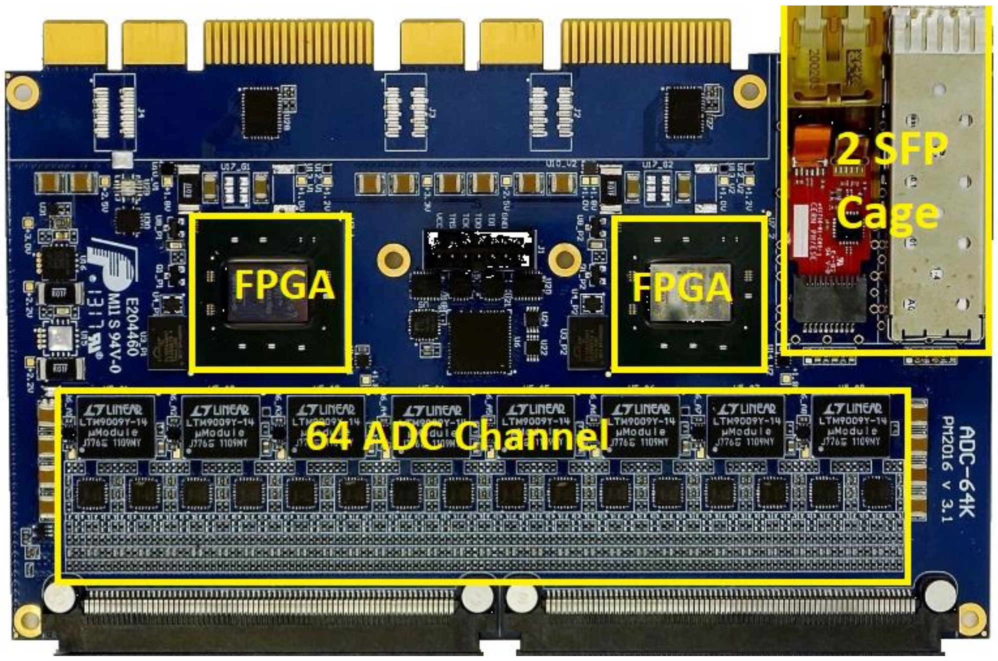


Figure 4- ADC board

2.3 Xilinx Zynq-7000

Before delving into the design, a brief description about the Zynq-7000 family architecture is given.

Zynq-7000 is one of the SOC family's FPGAs designed and produced by Xilinx that utilizes the ARM Cortex-A9. Zynq SOC, as the name suggests, is consisted of programming logic (PL) and processing system (PS), the former is usually the Kintex-7 or Artix-7 family FPGA and the latter is the processor core which can be a single or multi-core ARM CPU running at different frequencies. As depicted in the Figure 5, from Z-7010 Zynq class the device has the following hardware architecture and the ZynqBoard chip is in this family. [2]

In the Application Processor Unit (APU), it uses two Arm Cortex-A9 with 32-KB L1 Instruction and Data cache, 512 KB of L2 cache, ARM NEON engine which gives the CPU the Single Instruction Multiple Data (SIMD) capability and Floating Processor Unit (FPU) that inhance the CPU to offload Single/Precision calculations. With Snoop Control Unit (SCU) the CPUs can handle and manage cache coherency. In addition, the SOC has a 256 KB on Chip Memory (OCM) which can be used as a scratchpad. It also has a Memory Interface to connect to DDR2/3, and SRAM memories. The Cores utilize Memory Management Unit (MMU) which will be explained in detail later, but in short, it handles memory attributes and features in different memory domains and blocks, based on the cache coherency and access permissions. Moreover, the core has array of I/O peripherals such as Gigabit Ethernet, SPI, UART, USB, I2C, CAN, GPIO. [3]

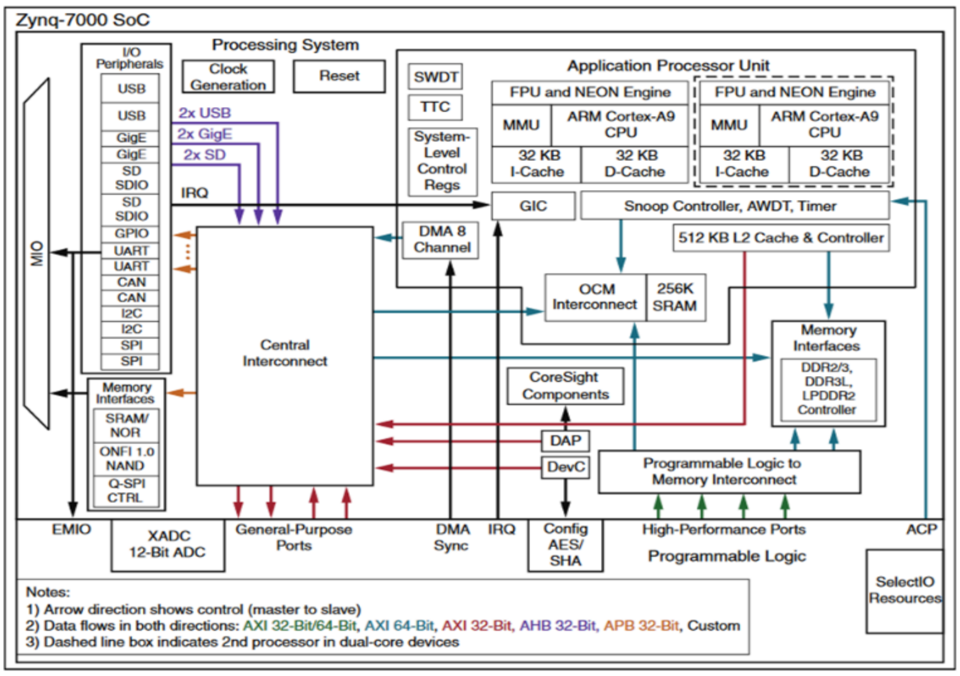


Figure 5 - Zynq-7000 SOC architecture [3]

2.3.1 Zynq Internal PS and inner PS-PL communication interfaces

Zynq Arm core for internal and inner PL and PS connection uses an Advanced Microcontroller Bus Architecture (AMBA) switch and AMBA Advanced Extensible Interface (AXI) protocol. [3] .

In all well-known FPGA SOC architecture AMBA AXI protocol is used and this prevalent standard protocol is not only used by ARM SOC FPGAs, but also is being used by other processors and soft processor architectures, like Xilinx Microblaze. AXI protocol has a variety of specifications with which it can handle all applications.

Having one standard protocol, reduces the complexity of the Intellectual Property (IP) design, offers IP portability and introduces the new capability that makes possible to use the third-party designs beside Vivado IP Catalog. [4]

The AXI protocol has the following features: [5]

a) Suitable for high-bandwidth and low-latency designs
b) Provides high-frequency operation without using complex bridges
c) Meets the interface requirements of a wide range of components
d) Suitable for memory controllers with high initial access latency
e) Provides flexibility in the implementation of interconnect architectures
f) Backward-compatible with existing AHB and APB interfaces.
g) Separate address/control and data phases
h) Supports for unaligned data transfers, using byte strobes
i) Uses burst-based transactions with only the start address issued
j) Separate read and write data channels, that can provide low-cost Direct Memory Access (DMA)
k) Support for issuing multiple outstanding addresses
l) Support for out-of-order transaction completion
m) Permits easy addition of register stages to provide timing closure
n) Allow different master and slave to be from different clock domain

2.3.1.1 Types of AXI interface

a) AXI

AXI interface is an addressable multi-master and multi-slave interface with very high throughput, bursting up to 256 data transfers which can have a data-width of 8, 16, 32, 64, 128, 256, 512, or 1024 bits.

The AXI interface has the following independent transaction channels:

- Read address
- Read data
- Write address
- Write data
- Write response

As said earlier, the AXI interface is a multi-master and multi-slave protocol and when masters and slaves are connected to each other, it forms the AXI interconnect that in most systems can have one of the three following topologies based on the five channels described:

- Shared address and data buses
- Shared address busses and multiple data buses
- Multilayer, with multiple address and data buses.

Address bus usually uses a very low bandwidth compared to data bus and in most of applications, it can be shared between different components. Meanwhile, each component will have an individual data bus which provide parallel data transfer for optimized performance. [5]

b) AXI-Lite

The AXI-lite is an interface like AXI but it does not support burst transfer and transmits one packet at a time. In this interface, the data width can only be 32/64 bit.

AXI-lite is usually used for configuration purposes or transfering data with low throughput.

c) AXI-Stream

The AXI-Stream is a simple data-centric protocol with configurable data width which is a single unidirectional interface without address channel, in other words, it is a master slave peer-to-peer interface connection, with almost unlimited burst size with any integer number of data width.

As depicted in the Figure 6, the Zynq family for interconnecting between the PL and PS has the following interfaces:

a) Four High Performance (HP) ports:

HP port is a 32/64-bit data width AXI-3 interface. This interface directly connects the PL to DDR controller or OCM interconnect

b) Accelerator Cache coherent Port (ACP):

ACP is a 64-bit data width AXI-3 interface that directly connects the PL to snoop control unit with the same CPU address space mapping and automatically handles cache coherency of data between all levels of memory hierarchy (DDR, L1 and L2 cache), without software intervention. In other word, the programmer does not need to explicity and consciously flushes or invalides the cache.

c) General Purpose AXI Slaves and Masters

General purpose AXI masters' and slaves' buses, as the name implies, are for general-purpose usage in the PL and are not intended for high-throughput connectivity. These AXI-3 interfaces are 32-bit data width, can issue 8 reads and 8 writes in the master and slave. General purpose AXI slaves and masters are connected to PS corresponding interconnects. In the PS, There are two different AMBA interconnects that all Slave and Master I/O peripherals on the PS side are connected to and because of this the throughput of the these AXI buses can be affected by the speed and occupancy of the interconnects.

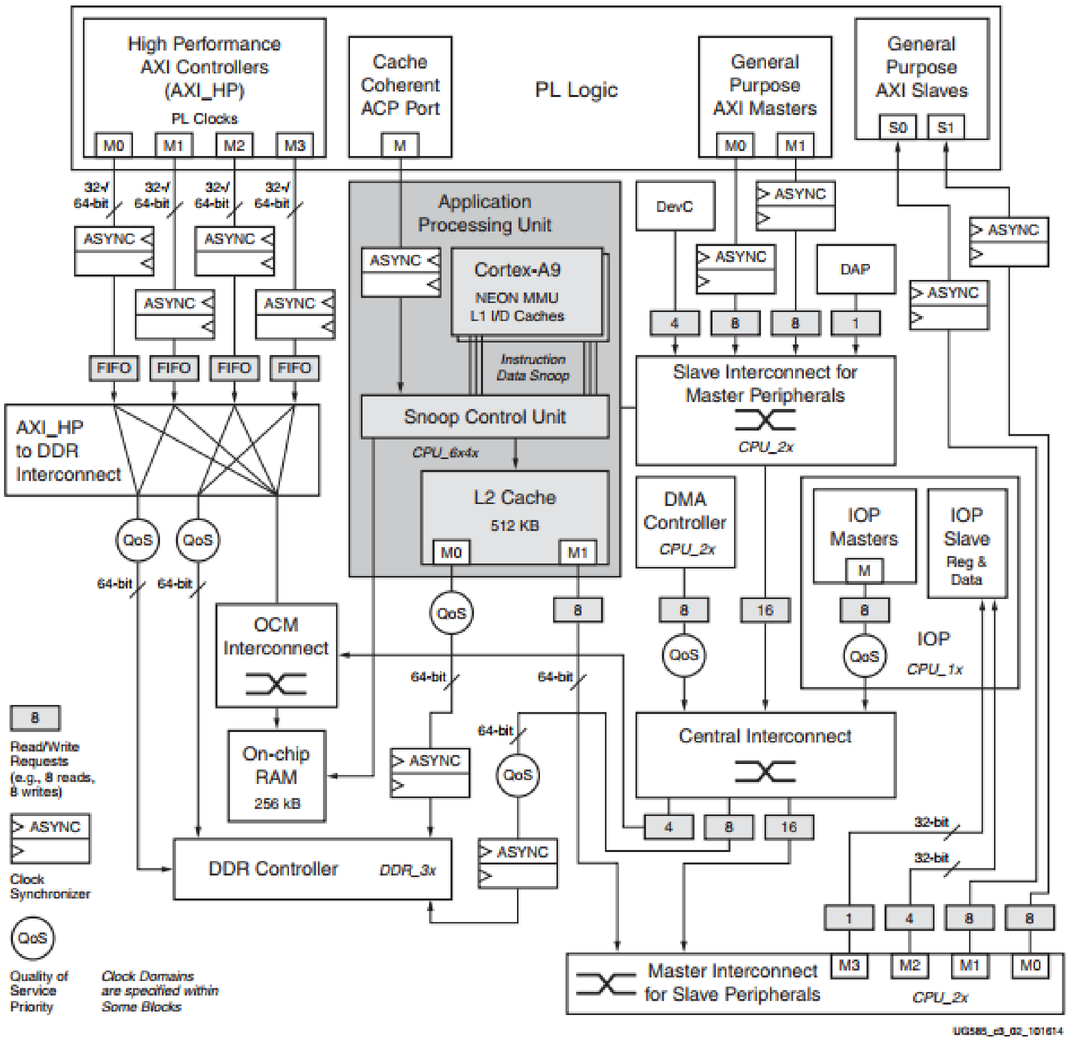
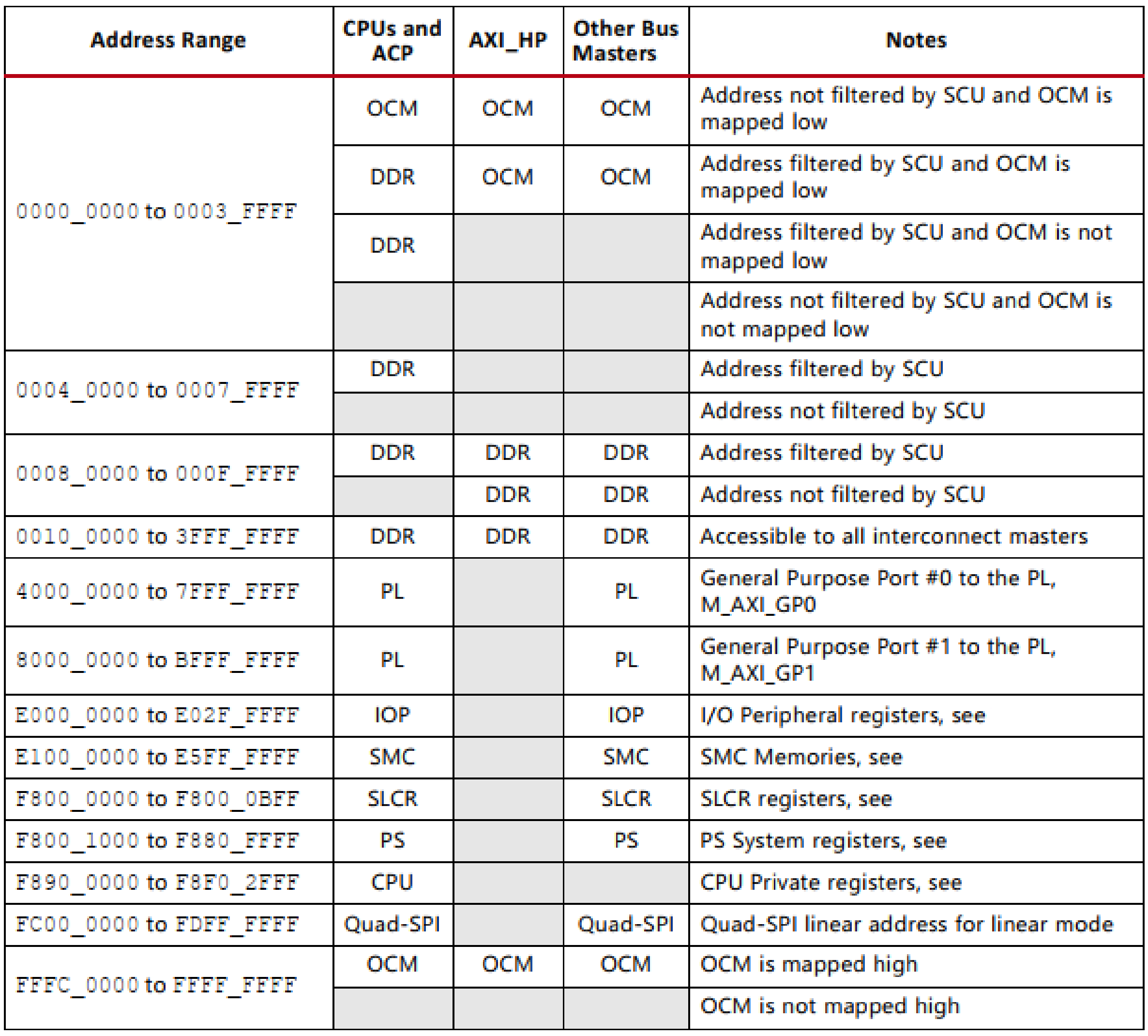


Figure 6-APU System View Diagram [3]

2.3.2 Memory

Zynq-7000 which utilizes the Cortex-A9 core(s) uses the ARMv7-A architecture. In this architecture, the memory address space model uses a single, flat address space of 2^{32} 8-bit, covering 4GBytes. The system level address map is shown in Table 1, based on this mapping the accessibility of the CPU and other main interfaces to different memory regions are specified. The shaded areas are reserved and cannot be used.

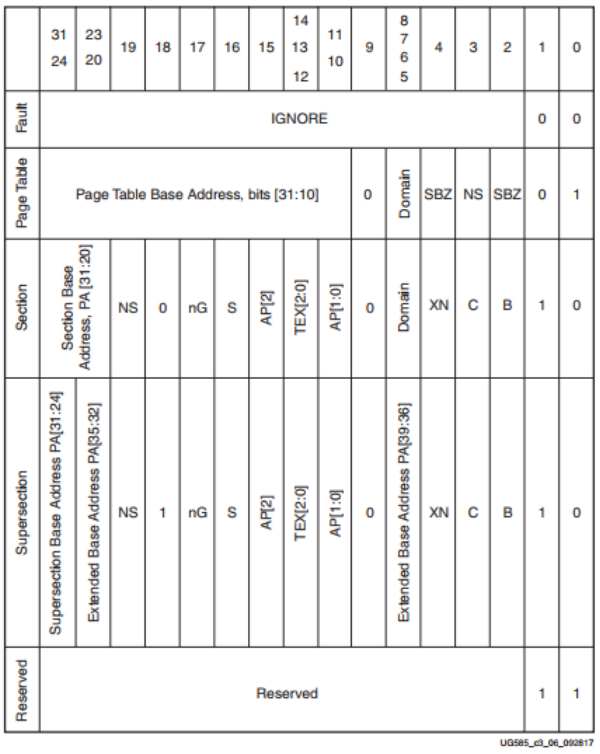
Table 1-System Level Address Map [3]



Address Range	CPUs and ACP	AXI_HP	Other Bus Masters	Notes
0000_0000 to 0003_FFFF	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see
F800_1000 to F880_FFFF	PS		PS	PS System registers, see
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see
FC00_0000 to FDFF_FFFF	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

The Memory Management Unit (MMU) handles memory translation and memory protection and is compatible with Virtual Memory System Architecture version 7 (VMSAv7) that requires 4KB, 64KB, 1MB page table entries with 16 domain tables. The key feature of the MMU is to do address translation; it translates the virtual address to physical address automatically and masks the complexity of memory access by the application, operating system or other running tasks. MMU uses table-based address translation scheme and for each cache, it uses an independent page table via which the MMU translates the address and this process is known as page table walking. As mentioned earlier, the core has a hierarchy of memory with two level data caches, 32KB L1 cache and 512KB L2 cache. The L1 page table which sometimes is called as master page table can work in four modes and typically divides the whole 4GByte address space into 4,096 equally 1MB sections. Each of 1MB sections has a page table entry which is shown in Table 2 and besides address transition, this table entry specifies the memory attributes. In most common mode, the section and these attributes can be modified via the operating system or application in bare-metal

Table 2- L1 Page Table Entry Format [3]



	31 24	23 20	19	18	17	16	15	14 13 12	11 10	9	8 7 6 5	4	3	2	1	0
Fault	IGNORE														0	0
Page Table	Page Table Base Address, bits [31:10]									0	Domain	SBZ	NS	SBZ	0	1
Section	Section Base Address, PA [31:20]		NS	0	nG	S	AP[2]	TEX[2:0]	AP[1:0]	0	Domain	XN	C	B	1	0
Supersection	Supersection Base Address PA[31:24]	Extended Base Address PA[35:32]	NS	1	nG	S	AP[2]	TEX[2:0]	AP[1:0]	0	Extended Base Address PA[39:36]	XN	C	B	1	0
Reserved	Reserved														1	1

UG585_c3_06_092817

.

The L1 page table entry has the following main configurations

a) Cacheability of the section (C)
b) Bufferability of the section (B)
c) Memory region attribute TEX

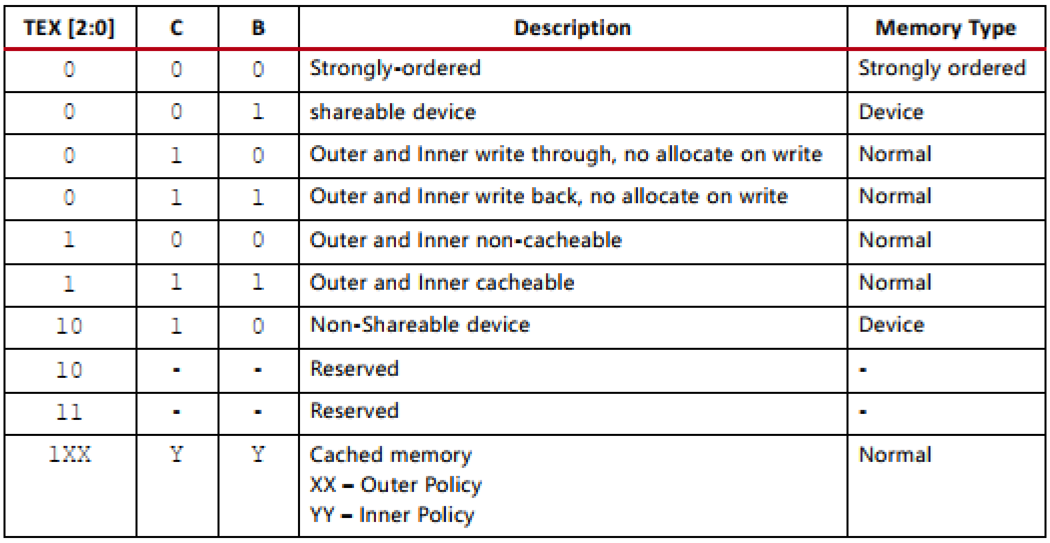
As shown in Table 3 and Table 4, with configuration and setting of the TEX, Cacheability and Bufferability bits, the expected memory section cache policy can be specified. In general via this settings three major cache policy can be achieved.

- Automatic cache update via write-though and write-back policies
- Strong orderly policy for the new hardwares that are customed and introced to the system
- Completely disabling cache

d) Access Permission(AP)

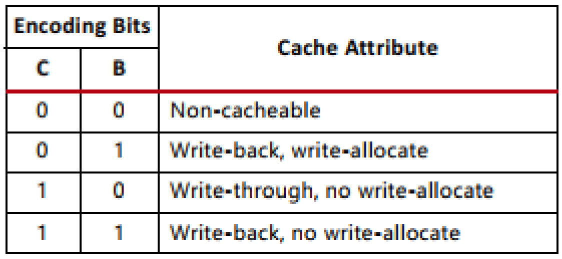
In this architecture, the memory can be categorized by different domains and each domain accessability, as it can be seen in Table 5, can be specified via access permission bits.

Table 4- Memory Attribute Encodings [3]



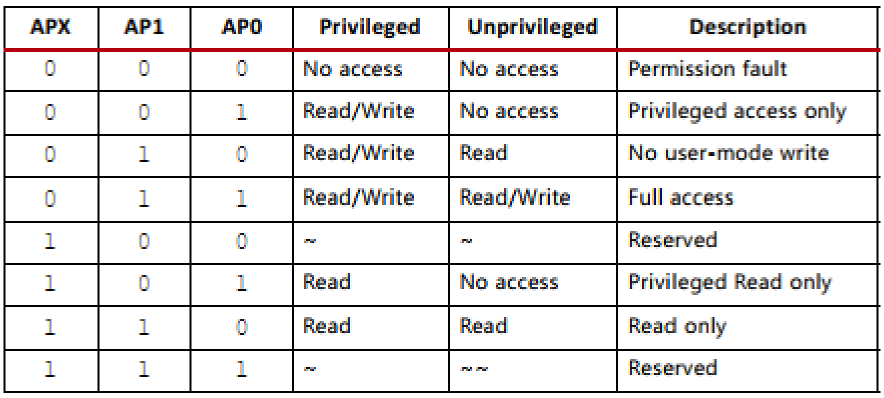
TEX [2:0]	C	B	Description	Memory Type
0	0	0	Strongly-ordered	Strongly ordered
0	0	1	shareable device	Device
0	1	0	Outer and Inner write through, no allocate on write	Normal
0	1	1	Outer and Inner write back, no allocate on write	Normal
1	0	0	Outer and Inner non-cacheable	Normal
1	1	1	Outer and Inner cacheable	Normal
10	1	0	Non-Shareable device	Device
10	-	-	Reserved	-
11	-	-	Reserved	-
1XX	Y	Y	Cached memory XX – Outer Policy YY – Inner Policy	Normal

Table 3- Memory Attributes Encodings [3]



Encoding Bits		Cache Attribute
C	B	
0	0	Non-cacheable
0	1	Write-back, write-allocate
1	0	Write-through, no write-allocate
1	1	Write-back, no write-allocate

Table 5-Access Permissions Encodings [3]



APX	AP1	AP0	Privileged	Unprivileged	Description
0	0	0	No access	No access	Permission fault
0	0	1	Read/Write	No access	Privileged access only
0	1	0	Read/Write	Read	No user-mode write
0	1	1	Read/Write	Read/Write	Full access
1	0	0	~	~	Reserved
1	0	1	Read	No access	Privileged Read only
1	1	0	Read	Read	Read only
1	1	1	~	~~	Reserved

In this project, the interest is on the memory attributes encoding. Changing the memory attributes for the selected region with SCU configurations play it’s rule when the ACP port for transmission of data is used to have automatic memory coherency.

2.3.2.1 OCM

The Cortex-A9 has On Chip Memory (OCM) module that contains two 256KB RAM. As demonstrated in Figure 6, the OCM is not L2 cacheable but has the following key features:

- Supports full AXI 64-bit bandwidth of simultaneous read and write commands.
- Low latency path for reads from CPU/ACP
- Supports random access to RAM for AXI masters

The OCM as depicted in Figure 7, has two main paths to be connected to, one from SCU and the other one via OCM Interconnect. In this project, based on the different interconnects, the OCM RAM connections will be explored.

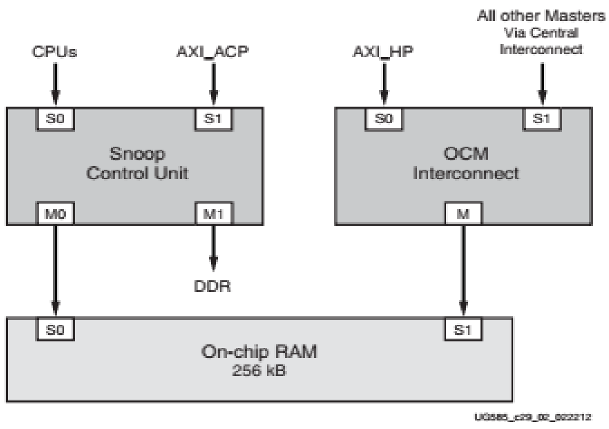


Figure 7-OCM System Viewpoint [3]

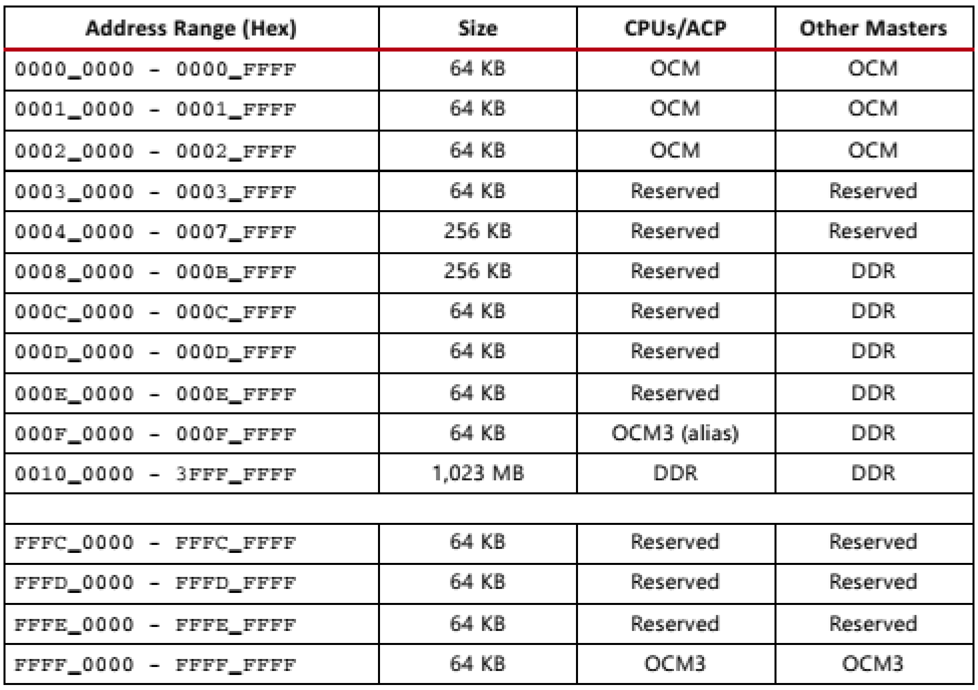
The OCM consists of four 64KB individual blocks with different address ranges. As it can be seen in Table 6, the default initial configuration of the OCM/DDR address is mapped in a way that three parts of OCM are visible to masters via lower addresses and OCM3 is accessible via higher address.

In some cases, as shown in Table 7, it is demanded to relocate the OCM memory address mapping in such a way that all the OCM parts can be visible to higher address. For having such a memory mapping, the following minimum changes must be made:

- slcr.OCM_CFG[RAM_HI] =1111
- mpcore.SCU_CONTROL_REGISTER[address_filtering_enable]=1
- mpcore.Filtering_Start_Address_Register =0x0000_0000
- mpcore.Filtering_End_Address_Register =0xFFE0_0000

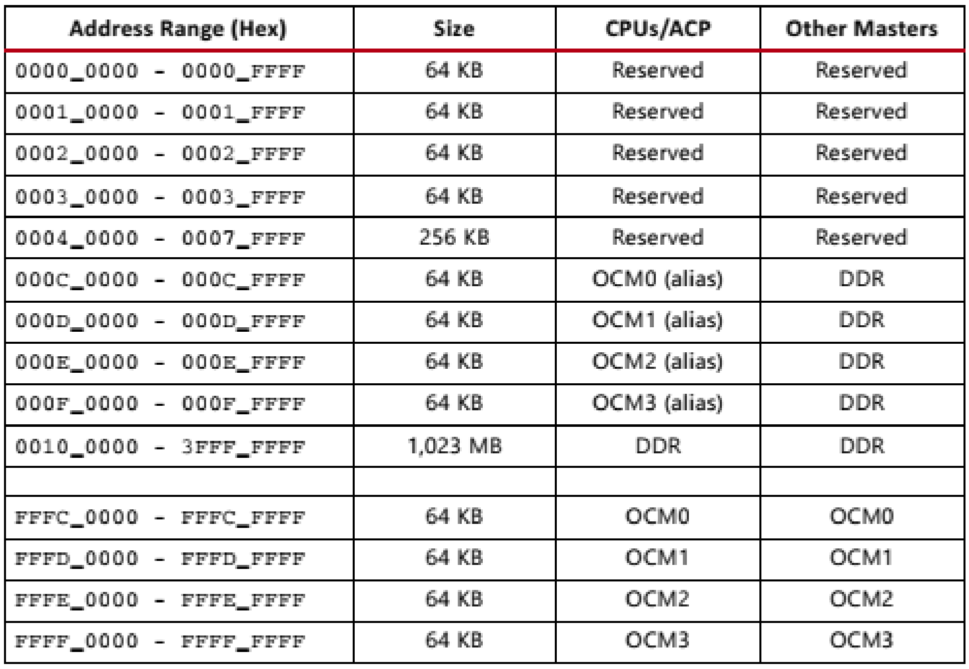
- In the Xilinx processing system configuration “allow access to High OCM” must be checked that masters on HP or ACP can access to OCM otherwise only CPU can access the High region

Table 6-Initial OCM/DDR Address Map [3]



Address Range (Hex)	Size	CPUs/ACP	Other Masters
0000_0000 - 0000_FFFF	64 KB	OCM	OCM
0001_0000 - 0001_FFFF	64 KB	OCM	OCM
0002_0000 - 0002_FFFF	64 KB	OCM	OCM
0003_0000 - 0003_FFFF	64 KB	Reserved	Reserved
0004_0000 - 0007_FFFF	256 KB	Reserved	Reserved
0008_0000 - 000B_FFFF	256 KB	Reserved	DDR
000C_0000 - 000C_FFFF	64 KB	Reserved	DDR
000D_0000 - 000D_FFFF	64 KB	Reserved	DDR
000E_0000 - 000E_FFFF	64 KB	Reserved	DDR
000F_0000 - 000F_FFFF	64 KB	OCM3 (alias)	DDR
0010_0000 - 3FFF_FFFF	1,023 MB	DDR	DDR
FFFC_0000 - FFFC_FFFF	64 KB	Reserved	Reserved
FFFD_0000 - FFFD_FFFF	64 KB	Reserved	Reserved
FFFE_0000 - FFFE_FFFF	64 KB	Reserved	Reserved
FFFF_0000 - FFFF_FFFF	64 KB	OCM3	OCM3

Table 7-OCM Relocation Address Map [3]



Address Range (Hex)	Size	CPUs/ACP	Other Masters
0000_0000 - 0000_FFFF	64 KB	Reserved	Reserved
0001_0000 - 0001_FFFF	64 KB	Reserved	Reserved
0002_0000 - 0002_FFFF	64 KB	Reserved	Reserved
0003_0000 - 0003_FFFF	64 KB	Reserved	Reserved
0004_0000 - 0007_FFFF	256 KB	Reserved	Reserved
000C_0000 - 000C_FFFF	64 KB	OCM0 (alias)	DDR
000D_0000 - 000D_FFFF	64 KB	OCM1 (alias)	DDR
000E_0000 - 000E_FFFF	64 KB	OCM2 (alias)	DDR
000F_0000 - 000F_FFFF	64 KB	OCM3 (alias)	DDR
0010_0000 - 3FFF_FFFF	1,023 MB	DDR	DDR
FFFC_0000 - FFFC_FFFF	64 KB	OCM0	OCM0
FFFD_0000 - FFFD_FFFF	64 KB	OCM1	OCM1
FFFE_0000 - FFFE_FFFF	64 KB	OCM2	OCM2
FFFF_0000 - FFFF_FFFF	64 KB	OCM3	OCM3

2.3.2.2 DDR

The PS has a DDR memory controller supporting variety of DDR memory types. Memory controller comprises of three major blocks: DDR Controller System Interface (DDRI), DDR Controller PHY (DDRP) and DDR Controller Core and finally Transaction Scheduler. In this project, there is no need to delve into DDR controller details, but the DDR interface is important and as it can be seen in Figure 8, the DDR interface has 4 individual Synchronous 64-bit AXI ports with separate read/write. In Zynq-7000 Technical Reference Manual [3] it is not explicitly mentioned that DDR controller is capable to handle simultaneous read and write like OCM, but is specified that DDR core, which handles management and scheduling, tries to do efficient transaction scheduling that optimizes the data bandwidth and latency. In this project the OCM and DDR memory via different ports and in different configurations will be explored.

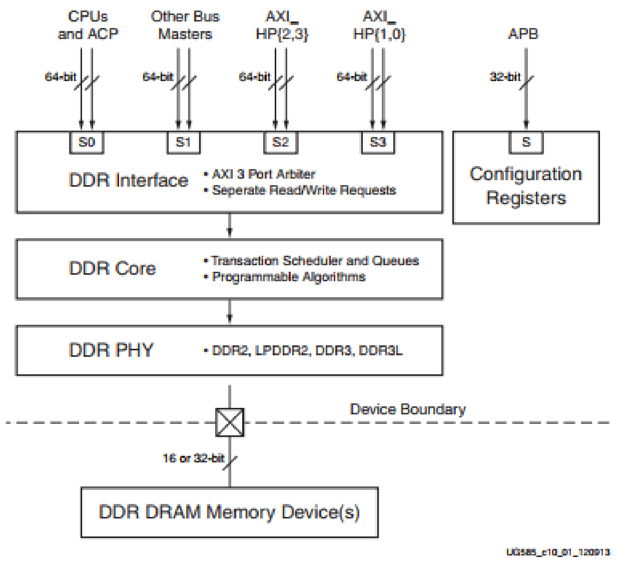


Figure 8-DDR Memory Controller Block Diagram [3]

3 Design

In this section the philosophy and design of the project is explained.

3.1 Overall view

As shown in Figure 9, the high-speed serial transceivers (GTX) are connected to SFPs via a differential voltage mode driver to collect the high-speed data stream from the ADC boards. On the other part of the transceiver, payloads are deserialized and converted to a low clock speed in 32-bit data format and passed to busy section. The busy section will handle the FIFO insertions while monitoring the status of FIFO. If busy section notices that the FIFO is about to be filled, it will send the BUSY signal to the sender ADC board to maintain the flow control. Since, each ADC board is sending its sampling data with its own clock, it is apparent that on the receiving part, the data should be handled with the recovered clock[2]. In other words, busy FSM and FIFO insertion logic is clocking by the connected sender ADC board recovered clock. Later, the Fetch FSM sequentially looks through non-empty FIFOs with specific threshold based on the protocol and fetches one complete payload from header to footer and transfers it to the bigger common FIFO. M_AXI_S is an FSM which based on the DMA request will read data from the common FIFO and send the data to PS via AXI stream protocol. The AXI protocol is chosen among the AXI family due to having the best performance and simplicity.

In addition, the burst size of the AXI stream is dynamic and is configurable via AXI lite link by PS. On the PS side, the data from PL will be transferred to memory via introduced AXI ports. The collected data in the memory is transmitted with PS application via TCP protocol to Linux host for storage.

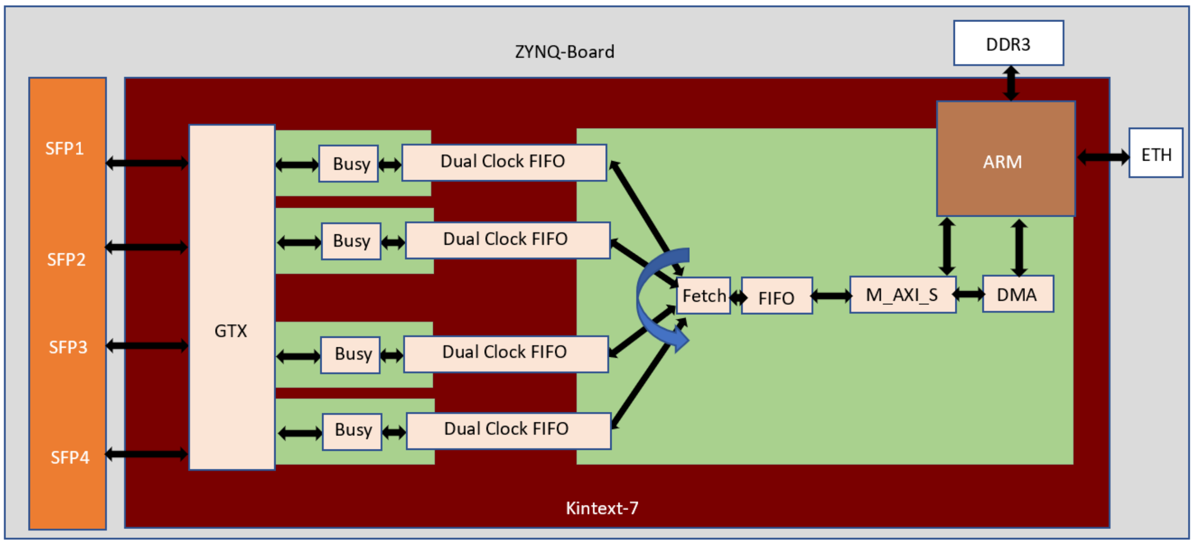


Figure 9-PL General Design (green sections are different clock domains)

[2] To maintain clock recovery, elastic buffers in high-speed transceivers are handy

3.2 PL logic

To reach the aforementioned functionality as described in the overall section, the logic is developed in the VHDL language, but before zeroing in on the logic, a context is given about the two main components of the design, the high-speed transceivers and FIFOs which are Xilinx catalog logiCore.

3.2.1 7 Series FPGAs GTX/GTH Transceivers

Some Xilinx FPGAs are enhanced with dedicated high-speed transceivers which by speed and features can be generally categorized in GTP, GTX, GTH, and GTZ groups. As depicted in Figure 10, each of 7 series Xilinx technologies utilize different type of transceiver and in this project, the FPGAs are Kintex technology and they can run up to 6.6 Gbps. [6]

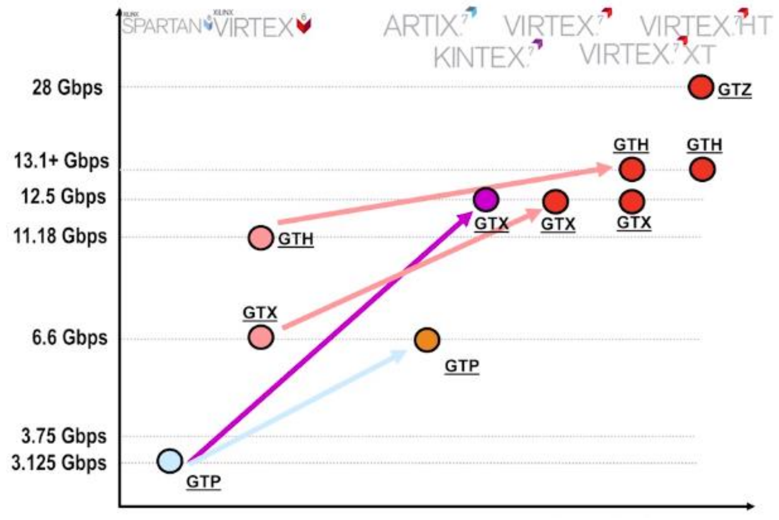


Figure 10-GT series speed vs technology diagram [22]

The high-speed transceiver is a dedicated hardware featured in specific FPGA packages and the transceiver is presented in the quad blocks. As shown in Figure 11, each quad block has four channels PLL and one quad channel PLL. Each channel PLL can be used individually which consumes more power compared to QPLL and CPLL is usable for speeds up to 6.6Gbps. For speeds higher than 6.6 Gbps, it is essential to use QPLL. [7]

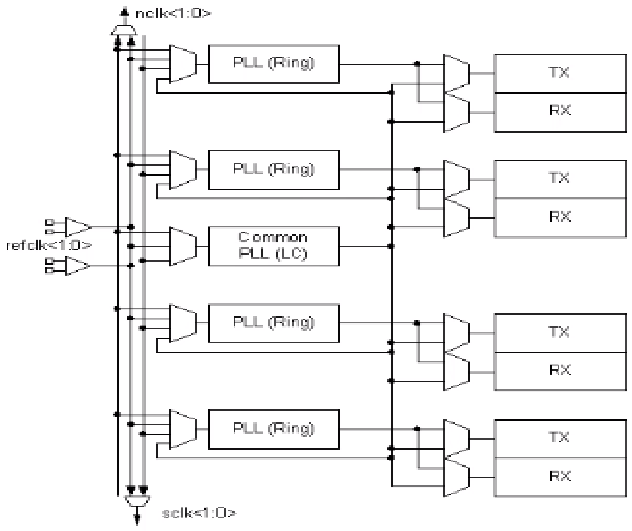


Figure 11-Transceiver Quad block [22]

The high-speed transceiver as shown in Figure 12, is consisted of two parts, Physical Media attachment (PMA) and Physical Coding Sublayer (PCS). PMA is responsible for converting the parallel data to serial format and connecting to media based on the required signaling, but in the majority of cases it is in Current Mode Logic (CML) to reduce the noise. PMA runs with the high-speed clock provided by channel PLL or Quad PLL. On the other hand, PCS which is responsible for preparing the data for transmission by PMA, runs with lower frequency. PCS is connected to the FPGA fabric in parallel 32, 64, and 128-bit data width and connects to PMA with 32- or 40-bit data width. It has a built-in 8b/10b decoder by which not only it sends data, but also does clock recovery at the receiver. [7]

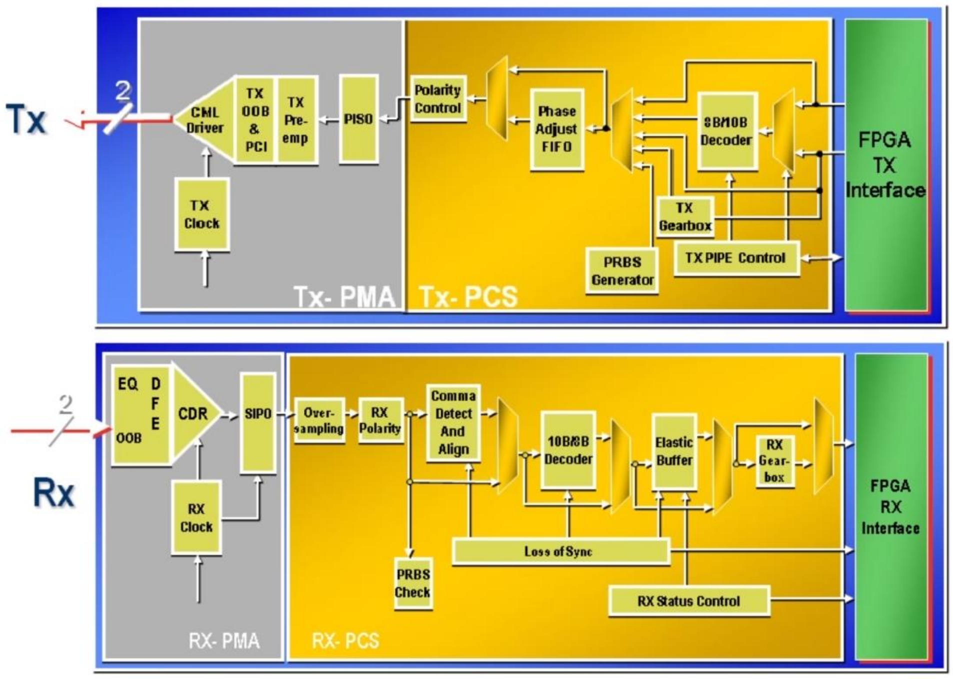
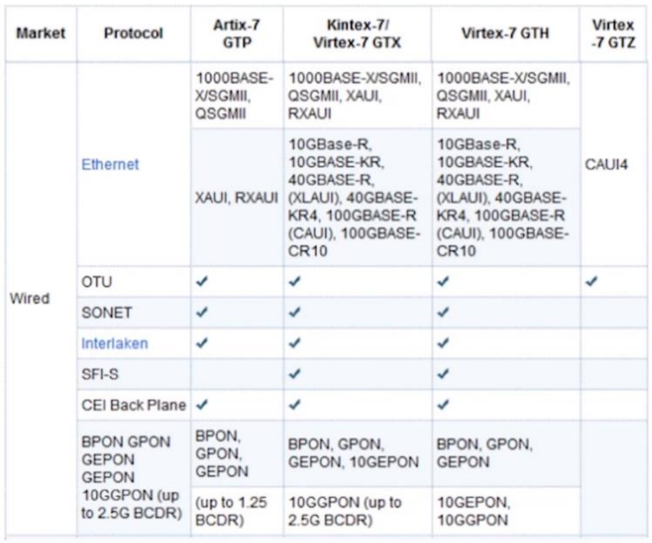


Figure 12- RX-TX subsection diagram [22]

The 7 series high-speed transceivers as shown in Table 8, supports variety of protocols but here in this project the ADC board uses a custom protocol specified as Table 9. The high-speed transceiver specification at the ZynqBoard should be configured the same way as ADC board to receive data correctly.

Table 8- 7 series Transceiver supported protocols [22]



Market	Protocol	Artix-7 GTP	Kintex-7/ Virtex-7 GTX	Virtex-7 GTH	Virtex -7 GTZ
Wired	Ethernet	1000BASE-X/SGMII, QSGMII	1000BASE-X/SGMII, QSGMII, XAUI, RXAUI	1000BASE-X/SGMII, QSGMII, XAUI, RXAUI	CAUI4
		XAUI, RXAUI	10GBase-R, 10GBASE-KR, 40GBASE-R, (XLAUI), 40GBASE-KR4, 100GBASE-R (CAUI), 100GBASE-CR10	10GBase-R, 10GBASE-KR, 40GBASE-R, (XLAUI), 40GBASE-KR4, 100GBASE-R (CAUI), 100GBASE-CR10	
	OTU	✓	✓	✓	✓
	SONET	✓	✓	✓	
	Interlaken	✓	✓	✓	
	SFI-S		✓	✓	
	CEI Back Plane	✓	✓	✓	
	BPON GPON GEPON GEPON 10GGPON (up to 2.5G BCDR)	BPON, GPON, GEPON	BPON, GPON, GEPON, 10GEPON	BPON, GPON, GEPON	
		(up to 1.25 BCDR)	10GGPON (up to 2.5G BCDR)	10GEPON, 10GGPON	

In addition, the ADC board sends the data in the format and segments as shown in Figure 13. Each payload is 276 Bytes. Consisting of header, channel number, collected data and the terminating footer. To maintain the flow control, to pause the sender ADC board from sending data when the FIFO is full, the receiver sends the 0x0001000 word every 8 cycles.

Table 9- ADC transceiver specification

Item	Config	Rate
1	Transceiver speed	2Gbps
2	Encoding	8b/10b
3	PCS internal data width	40-bit
4	TX-RX buffers for synchronization	
5	Comma value	K 28.5
6	Comma alignment	Four Byte Boundaries
7	Four-byte Clock correction sequence	000000BC
8	K character	BC

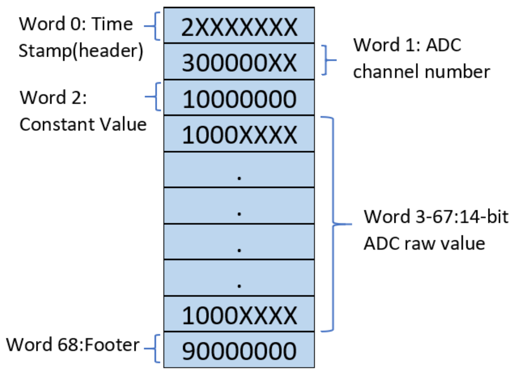


Figure 13-Payload Data Format

3.2.2 FIFO

Xilinx in its vast catalog repository has FIFO generator IP. With this generator, it is possible to generate the appropriate FIFO based on need. In its wizard, the width of the Input and the Output, dual clock read write mode, the depth of FIFO, the almost full threshold, and especially first word fall through mode can be specified. In the first word fall through mode, without issuing the read, the first inserted data will sit in the output pins. In FSM design this feature will help the design significantly.

In this project, two different FIFOs are needed. From each high-speed transceiver, the data are inserted to input FIFOs and the other FIFO is the common FIFO which accumulates all the data from all input FIFOS before sending the collected data to PS via AXI Stream.

The input buffer needs to have two separate clocks. As shown in Figure 14, for writing to FIFO it runs with the recovered clock from high-speed transceiver. For reading from FIFO, the FIFO runs with FPGA clock source which in this project would be from PS running at 100Mhz. The FIFO should have a prog_full threshold signal notifying the busy section. The busy section not only sends the busy signal to ADC board in cycles, but also it handles the remaining data that are being sent from the ADC and the high-speed transceiver elastic buffer before pausing.

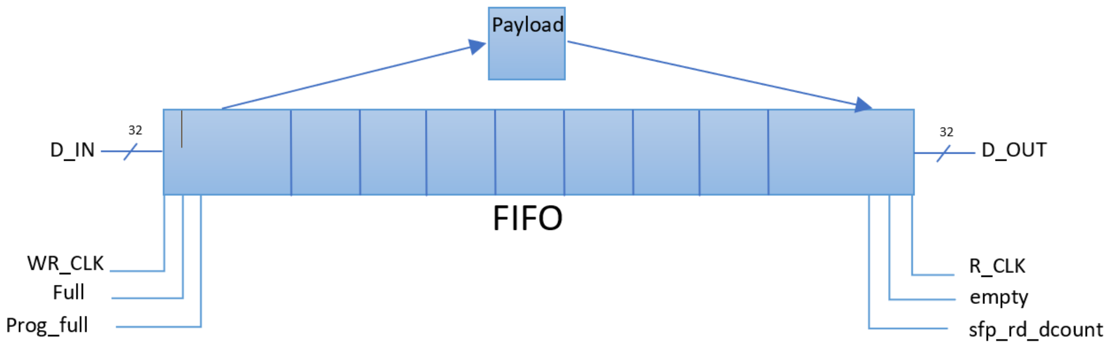


Figure 14- Input FIFO

In addition, the input FIFO has a read count port that shows the occupancy of the FIFO, which is being used by the Fetch FSM.

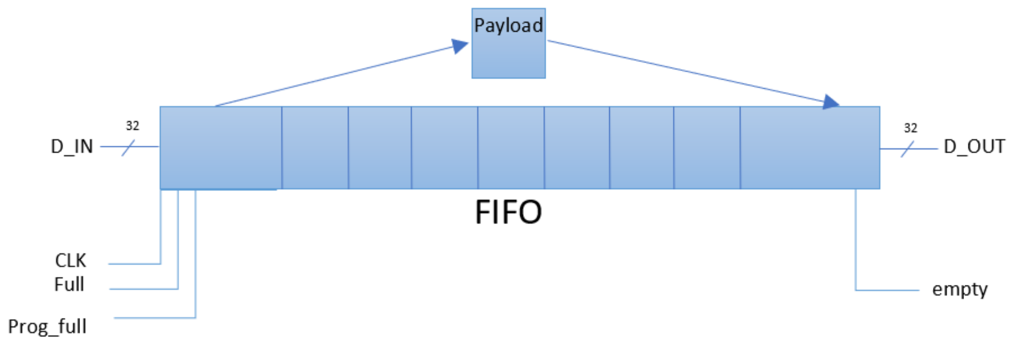


Figure 15-Common FIFO

On the other hand, as shown in Figure 15, the common FIFO has one clock for reading and writing and it does need to have prog_full signal based on the necessity of this project.

3.2.3 logic design

To reach the desired functionality of the PL logic the following three main sections with FIFO and high-speed receiver IPs must be implemented and connected together.

3.2.3.1 BUSY

The busy section, as the name suggests, handles the flow control and also via the fill FSM performs the data insertion. For handling the flow control, busy section monitors the FIFO threshold pin. When the FIFO reaches the threshold setpoint the busy signal (HEX "00001000") is sent to the sender ADC. But it takes some cycles to stop the expected ADC to cease the transmission, meanwhile the FIFO should have enough space available to handle the remaining data to maintain data consistency.

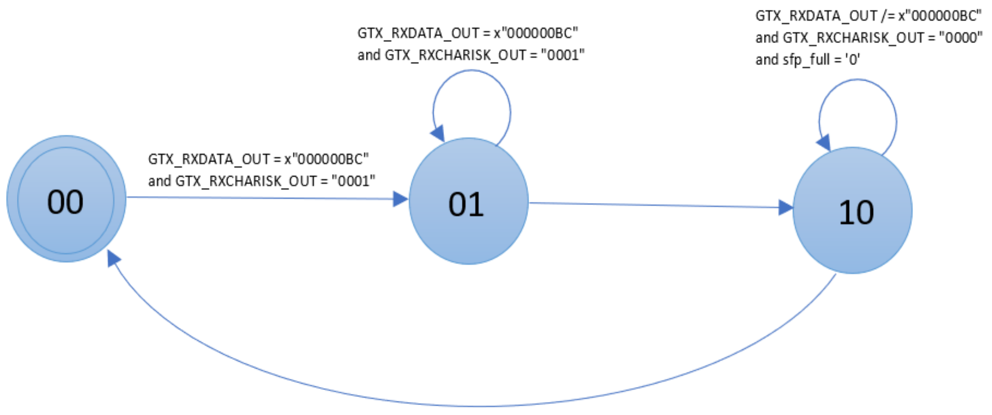

GTX_RXDATA_OUT = x"000000BC"
and GTX_RXCHARISK_OUT = "0001"
GTX_RXDATA_OUT /= x"000000BC"
and GTX_RXCHARISK_OUT = "0000"
and sfp_full = '0'
GTX_RXDATA_OUT = x"000000BC"
and GTX_RXCHARISK_OUT = "0001"
00
01
10

Figure 16-Fill FSM

The other part of the busy section is the fill FSM which, as shown in Figure 16, has three states. The FSM before data insertion checks for the appearance of at least two K characters in the received data sequence. The reason to check for two consecutive appearances of K character is to have a better deterministic behavior of the system.

3.2.3.2 FETH

Fetch section is an FSM which fetches the data from every FIFO in sequential order and puts them to the common FIFO. This Mechanism is implemented in five states. The machine starts in the “000” which is the IDLE state and sets the loop index to zero and if the common FIFO has enough capacity for one payload and the indexed input FIFO has more than 90 words of data, it will go to the next state, otherwise it will go to “100” state to change index by incrementing by one and start back from IDLE state. It is important that the threshold of prog_full is set in a way that in one fetch FSM execution, the common FIFO has a guaranteed space available for reading from one input FIFO. In this project, since each payload is 69 words, the threshold for the common FIFO is set to FIFO’s capacity-100.

In the “001” state, it will look for the header and since all FIFOs are set to first word fall through mode, there is a valid data on the indexed FIFO output. If the data on the FIFO is the header, it skips state to “011”, otherwise it goes to state “010” and it will search through the indexed FIFO to the depth of 20 looking for the header and if it does not find the header it will change state to “100” for incrementing the loop and going to IDLE state for fetching the next FIFO.

In the state “011”, it will fetch the data from the indexed FIFO till it finds the footer and inserts the fetched data to the common FIFO and when it finds the Footer it will switch back to “100” mode for changing the state and continuing looping through the FIFOs.

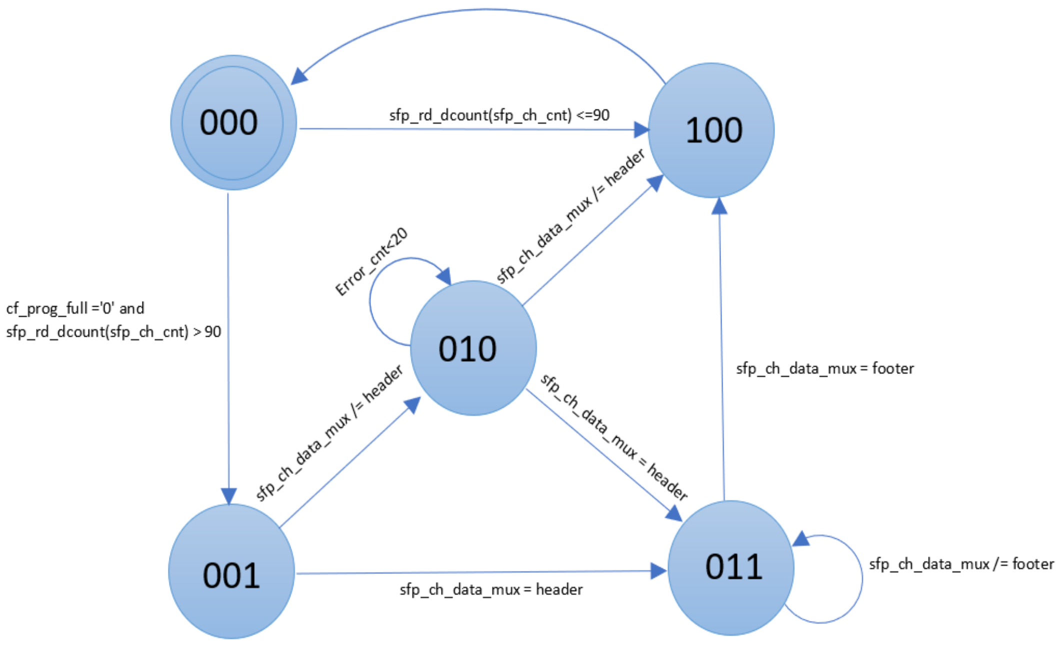

000
100
010
001
011
sfp_rd_dcount(sfp_ch_cnt) <=90
cf_prog_full ='0' and sfp_rd_dcount(sfp_ch_cnt) > 90
Error_cnt<20
sfp_ch_data_mux /= header
sfp_ch_data_mux /= header
sfp_ch_data_mux = header
sfp_ch_data_mux = footer
sfp_ch_data_mux /= footer
sfp_ch_data_mux = header

Figure 17-Fetch control FSM

3.2.3.3 AXI Master

This module has two AXI ports. AXI lite for controlling and setting the burst size and the AXI stream master for sending the data. The AXI stream has a signal named TLAST by which it notifies the recipient the last word in transmission. Asserting this signal at the right time is corresponded to burst size which should be configured via the AXI lite. The AXI lite, which was explained in the previous section, is the best bus for sending commands. In this module, for tuning purposes, it is important to set up the burst size and it is vital to be dynamic, otherwise by each modification the whole design must be synthesized and the new bitstream must be generated.

For setting up and developing the AXI lite in this module, the Xilinx template with four-registers slave is used and modified. Register zero is set as output to set the AXI stream burst size and the second register with sending 0x12340000 enabled the IP.

Upon a request from the DMA, the AXI stream master will fetch data from common FIFO and streams the data.

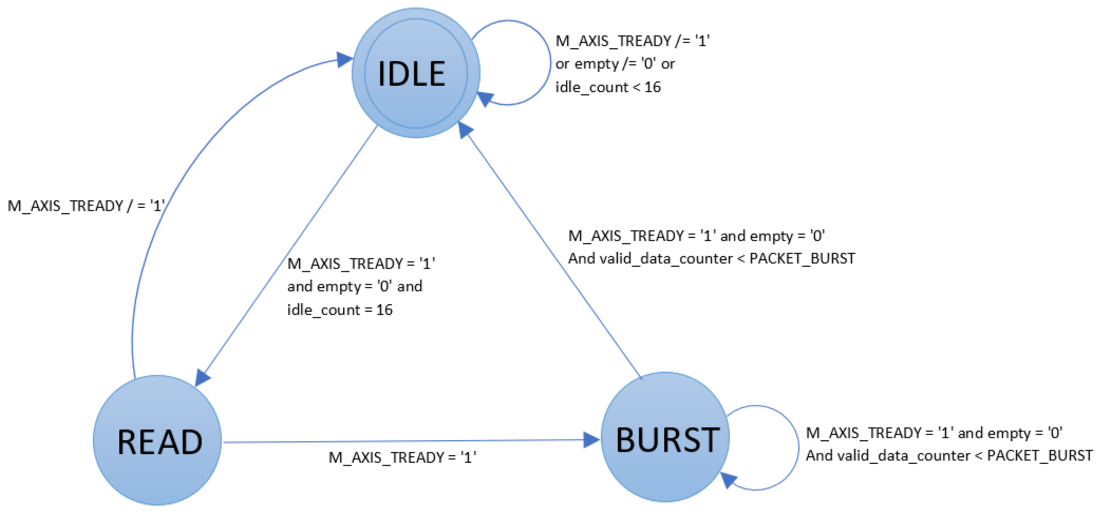


Figure 18-AXI STREAM MASTER FSM

As shown in Figure 18, the AXI Stream control is controlled by the finite state machine. The FSM is consistes of three states, machine is in the IDLE state while counting up to 16 for stability and if the DMA is not able to receive data or the common FIFO is empty it will stay in this state. Otherwise, If the DMA is ready to receive and the FIFO has some value in it, the machine changes state to read by asserting the common read signal. Meanwhile, if by any chance in this state the DMA is not ready to receive data, the machine will change state back to IDLE state. If the DMA has not changed state, the AXI Stream Master will change to BURST state and it will be in this state as long as the valid_data_count has not reached the burst size, the DMA is still accepting the data and the common FIFO is not empty. After this state, the machine changes state to IDLE.

In addition, valid_data_count is incremented by another process, in other words, while the FSM is streaming and it has an immature termination due to being interrupted by FIFO or DMA, the state machine switches back to IDLE and with having another process it will maintain its correctness.

Finally, the whole PL component must be packaged as custom IP and used along with the other design components in the block design canvas.

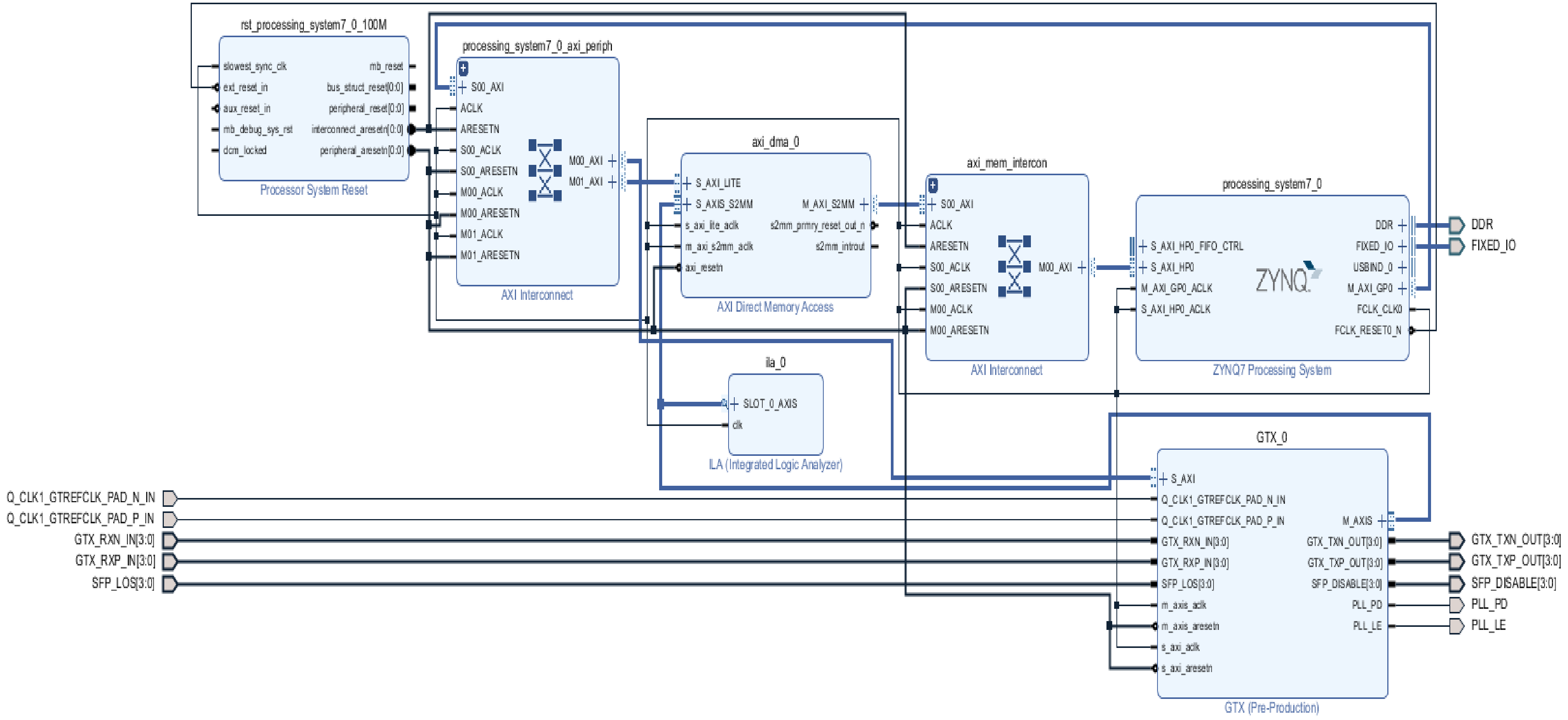


Figure 19- PL-PS block design

3.3 DataMover

As shown in Figure 20, the HP data path can be used to move data from PL to PS memory, which will cause non-coherency between two levels of the CPU caches. To maintain memory consistency in the memory hierarchy, explicit flushing and invalidation of the memory is needed by the programmer. Such acts, will have a huge cost to performance of the program. On the other hand, ACP port alongside with the SCU as a hardware logic can do the transfer and maintain the cache coherency in the background with some modification in L1 page table entry for specific region.

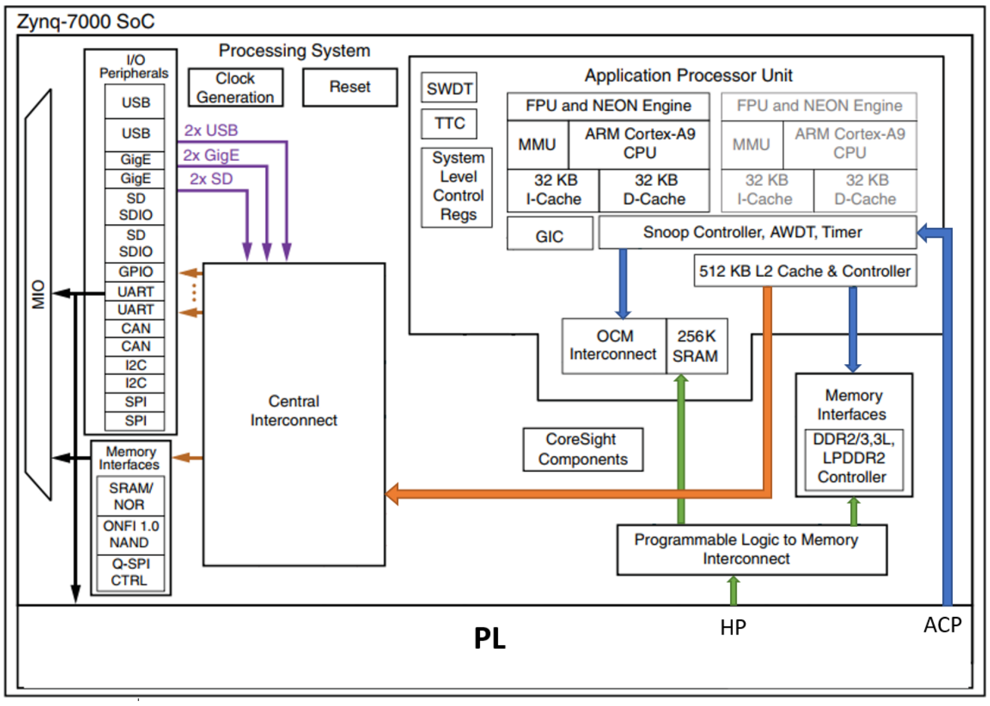


Figure 20- PS-PL data flow

As described in the introduction section shown in Table 6, the OCM consists of four 64KB memory sections. With default configuration, the first three memory sections are at the beginning of the Zynq memory map and the last section is at the high memory map and for consistent view it would be good to have all of them at the high address space. For achieving this in the ARM architecture, the XLSCR is unlocked and after setting the register OCM_CFG to 0x1F value, the XLSCR is locked. [3] Based on the Table 10, OCM is accessible to CPU, ACP and HP port.

Table 10- OCM high address

OCM number	**Address**
First block	0xFFFC0000
Second block	0xFFFD0000
Third block	0xFFFE0000
Fourth block	0xFFFF0000

In general, the OCM section must be set in a way that the cache coherency is maintained. Since the OCM does not go through the L2 cache, just modifying the L1 cache register in MMU for the corresponding OCM memory region is adequate.

As described in the introduction section, L1 cache breaks the whole 4 GB address space into 1MB sections, and for each section it has a table entry configuration. Since the whole OCM with new settings is visible at high address space, it somehow has consecutive address in the memory map. The whole address space of OCM can fit in less than 1MB. So, to achieve cache coherency altering one L1 entry table is enough and it can be achieved by setting the value for L1 TLB attribute with 0x10c06 value for the region starting with the address 0xfff00000.

If the DDR memory is chosen for transferring, flushing and cache invalidation is needed. But since in this project the PS is just reading the data, there is only need for invalidation. So, before any data movement from DMA to PS the recipient memory region specified by the DMA as the destination address must be invalidated.

For testing and exploring the ACP and HP port of the Zynq device, a simple counter IP as a data generator has been designed and by demand the counter sends consecutive data to the targeted memory via DMA. By using the counter any data miss and corruption can be easily be monitored and the correctness of the transmission can be evaluated.

In this project, to test the functionality of the design, As shown in Figure 21, in the block design canvas the following components have been used and put together:

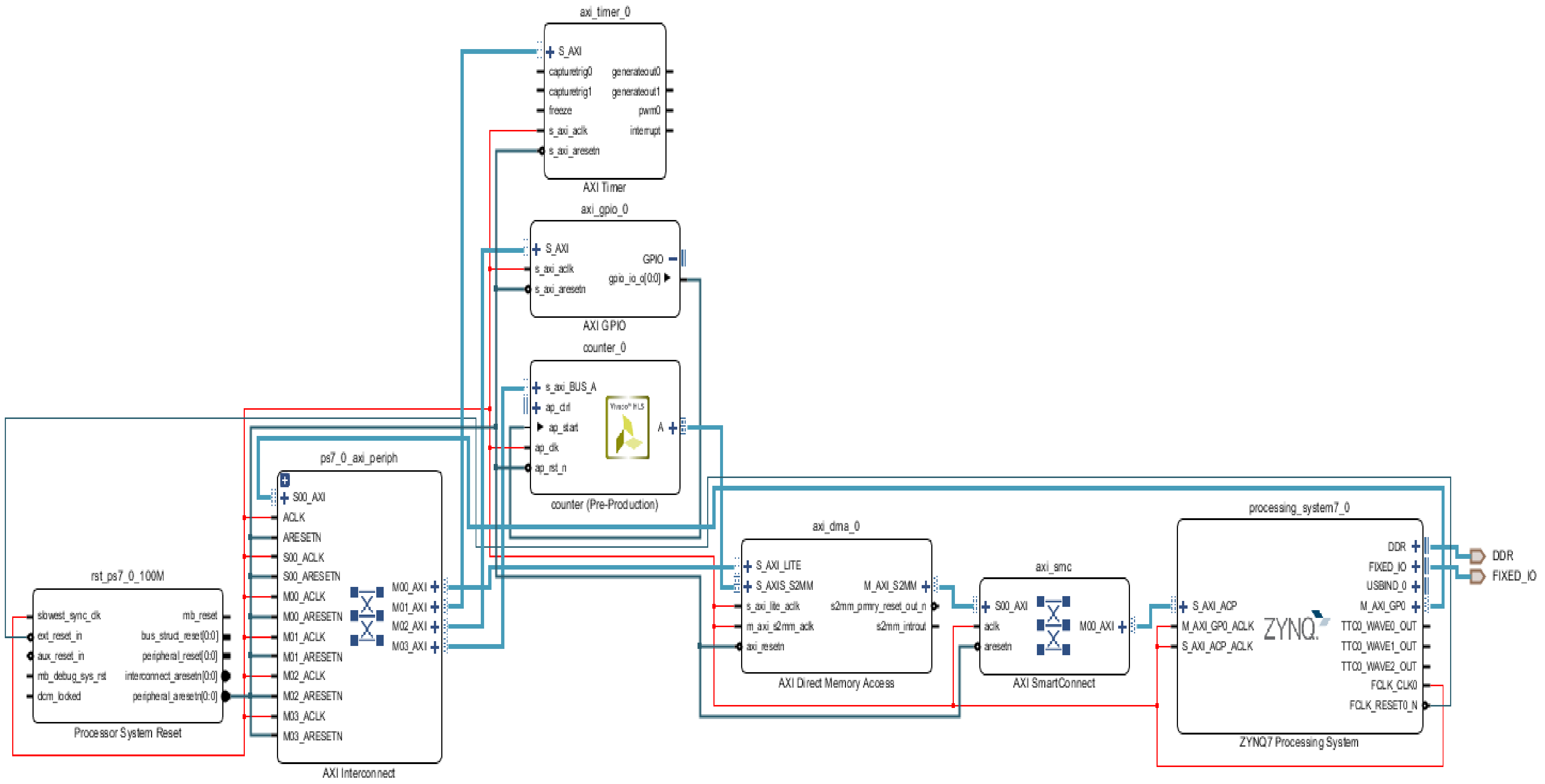


Figure 21-Block Design

3.3.1 ZYNQ7 Processing system

Zynq7 processing system IP, as shown in Figure 22, is a wrapper that instantiates the ARM core section of the SOC. In this section besides specifying all the communication interfaces of PS, which is the main rule of the wrapper, the wrapper also manages the peripherals of the PS and their behavior.

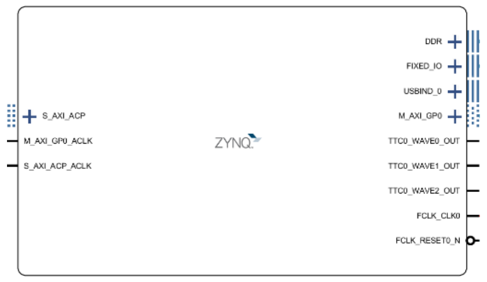


Figure 22-ZYNQ7 Processing System

The Zynq 7 Processing System as shown in Figure 23, has a configuration wizard. In this wizard, in the PS-PL Configuration tab for the purpose of this project the following changes are applied:

a) Based on the experiment either ACP or HP port is selected
b) Allow access to HIGH OCM is selected
c) Zynq can provide 4 independent PL clock sources and here one clock is set to 100MHz for PL components to be in sync with the PS

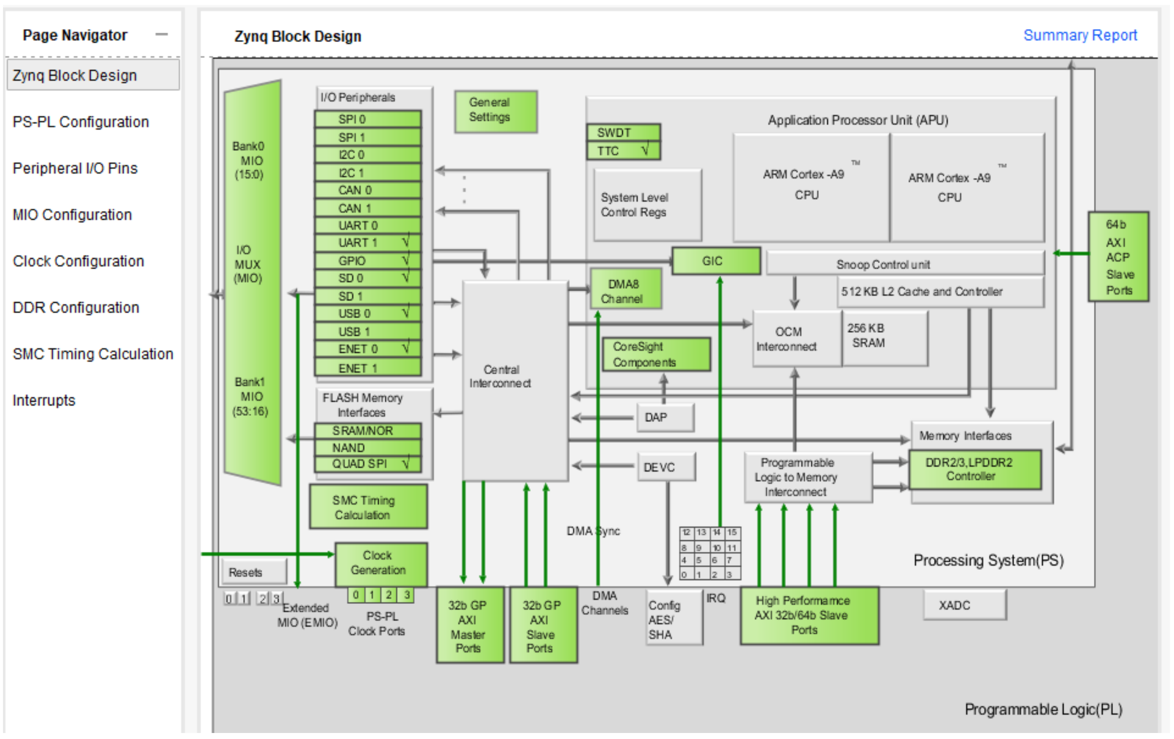


Figure 23-Zynq 7 Configuration Wizard

3.3.2 Counter

The Counter IP is designed in Vivado High Level Synthesis (HLS) and packaged as an IP. HLS has been chosen in this experiment instead of RTL for the following reasons [8]:

a. The design and test benching are performed in C/C++
b. All sort of interfaces can be handled by the HLS with simple direction configurations.
c. Based on the interface configurations a driver is automatically generated by the HLS that can be used in the SDK, either in bare-metal or Linux.
d. Design and upgrades can be performed in a very short time compared to the RTL design.

Counter Code:

```
#include <iostream>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

using namespace std;

typedef ap_axis <64,1,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;

void counter(STREAM_T &A,ap_uint<64> LEN,ap_uint<1> operand,ap_uint<64> start_value){

#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE s_axilite port=LEN         register bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=start_value register bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=operand     register bundle=BUS_A
AXI_T tmpA;
static unsigned long long loop=start_value;

  for(unsigned i=0; i<LEN; i++){

    tmpA.data=loop;

    if(i == LEN-1){
      tmpA.last = 1;
    }else{
      tmpA.last = 0;
    }
    tmpA.strb = 0xff;
    tmpA.keep = 0xff;
    A << tmpA;
    if(operand==1)
      loop++;
    else
      loop--;
  }
}
```

In the above code, the AXI_T which is an AXI stream configuration data type is defined, with specified predefined values. The AXI bus has 64-bit bus width and includes TLAST signal by which notifies the recipient of the last data word in transmission.

The whole counter IP is implemented in counter function with four arguments including the output data A, the LEN specifying the amount of data to be generated, start_value indicating the initial value of the counter, and finally, operand which specifies the operand ++ or –- to be performed on the start_value.

HLS has a smart compiler and with the help of the following pragma directives the behavior of the arguments are simply specified.

```
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE s_axilite port=LEN         register bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=start_value register bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=operand     register bundle=BUS_A
```

The first directive specifies that the A argument of the counter function is an AXI stream interface and automatically, based on the code structure, the compiler realizes that it is an output port and sets it as master.

The remainder directives specify that the remainder arguments are AXI-Lite specific and based on the bundle=BUS_A every argument is handled with the same AXI-lite bus. Furthermore, the arguments are set as registers to hold the data after module setup.

After function call, the start value is passed to static variable so that in next calls, the counter can continue with the last value to keep track of the data and maintain consistency in the experiment.

```
static unsigned long long loop=start_value;
```

In the main loop of the counter, TLAST signal is asserted based on the LEN configuration, and STRB which notifies the recipient about the validity part of the received packet is specified.

After exporting the counter as RTL, the IP will have an additional control signal, as it can be seen in Figure 24. It is important that ap_start signal be asserted otherwise the IP will not work as expected. The control part could be configured via AXI-Lite interface, but after each complete run of the counter IP, the IP needs to be reenabled, causing unnecessary CPU side cycles.

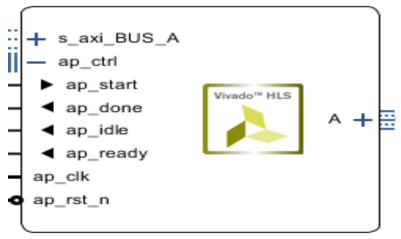


Figure 24- Counter IP

3.3.3 DMA

AXI DMA (7.1) as shown in Figure 25, is an IP core from the Vivado IP catalog capable of performing read and write from AXI to AXI stream operating in two modes. The first mode of operations is simple transfer; in this mode the DMA is instructed from an AXI-lite master, which could be a CPU to perform a transmission and after completion the DMA can interrupt the CPU or the status of the transmission can be polled by the CPU on demand. Second mode of operations is scatter gather; in this mode, either DMA can be given the larger set of transmission instructions to transfer like simple transfer mode or it can do cyclic transmission of data without any further CPU intervention to perform more complex transmissions. In this mode, instructions are stored in a BRAM which is programmed via master before DMA operation.

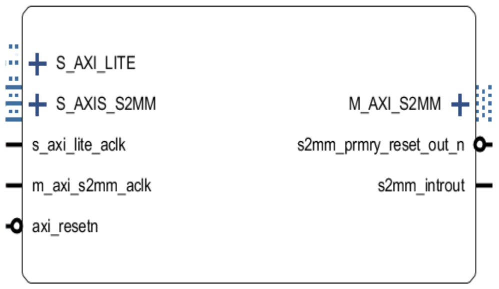


Figure 25-DMA

The key features of AXI DMA is as follows:

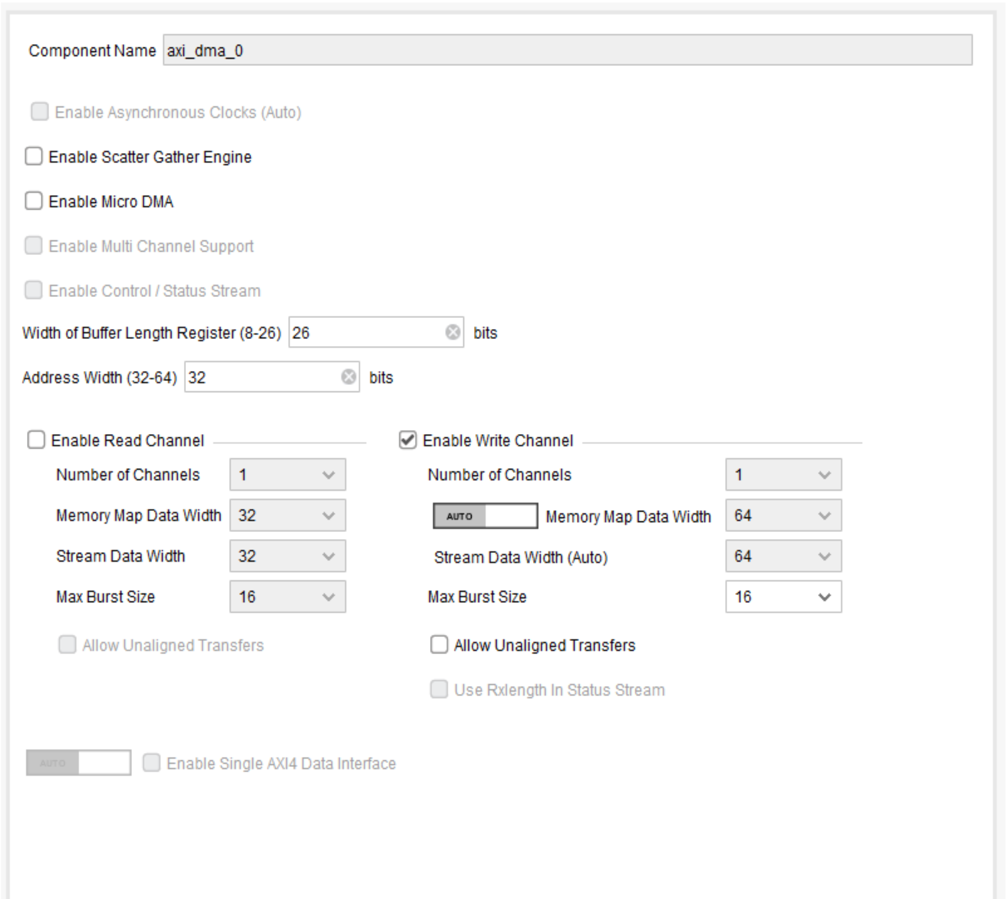
- AXI4 compliant
- Optional Scatter/Gather Direct Memory Access (DMA) support
- AXI4 data width support of 32, 64, 128, 256, 512 and 1,024 bits
- AXI4-Stream data width support of 8, 16, 32, 64, 128, 256, 512 and 1,024 bits

DMA 7.1 as shown in Table 11, can operate with the maximum frequency based on the device family type and the speed grade of the Zynq package and it must be considered before the design. [9]

Table 11-DMA Maximum Frequency [9]

Family	Speed Grade	Fmax (MHz)		
		AXI4	AXI4-Stream	AXI4-Lite
Virtex®-7	–1	200	200	180
Kintex®-7		200	200	180
Artix®-7		150	150	120
Virtex-7	–2	240	240	200
Kintex-7		240	240	200
Artix-7		180	180	140
Virtex-7	–3	280	280	220
Kintex-7		280	280	220
Artix-7		200	200	160

As shown in Figure 26, DMA IP core has a configuration wizard. In this project, the write channel is only selected since transmission is from PL to PS and the width of buffer length register is set to 26, which lets the DMA the ability to transfer 2^{26} Byte of data in one call. The burst size of write channel is set to 256, which is the maximum AXI-4 burst size. However, this modification does not have a significant performance improvement, since Zynq ports are AXI-3 and AXI-3 burst size is 16.



Component Name: axi_dma_0

- [ ] Enable Asynchronous Clocks (Auto)
- [ ] Enable Scatter Gather Engine
- [ ] Enable Micro DMA
- [ ] Enable Multi Channel Support
- [ ] Enable Control / Status Stream

Width of Buffer Length Register (8-26): 26 bits

Address Width (32-64): 32 bits

[ ] Enable Read Channel	[✓] Enable Write Channel
Number of Channels: 1	Number of Channels: 1
Memory Map Data Width: 32	AUTO Memory Map Data Width: 64
Stream Data Width: 32	Stream Data Width (Auto): 64
Max Burst Size: 16	Max Burst Size: 16
[ ] Allow Unaligned Transfers	[ ] Allow Unaligned Transfers
	[ ] Use Rxlength In Status Stream

AUTO [ ] Enable Single AXI4 Data Interface

Figure 26-DMA Configuration Wizard

3.3.4 Timer

AXI Timer is a simple 32/64-bit LogiCore Timer/Counter which interfaces through AXI-Lite. In this project, the timer is used to keep track of time in the PL to measure the transmission time. [10]

3.3.5 AXI GPIO

The AXI GPIO is a LogiCore Xilinx catalog that can be operated in a single or dual-channel mode with AXI-Lite interface. In this project, the AXI GPIO controls the behavior of the counter IP.

3.3.6 Processor System Reset

This module provides a customizable reset for the design which supports synchronous PS internal reset and external asynchronous reset.

3.3.7 AXI Interconnect

The AXI interconnect module core connects and converts different AXI versions and AXI types together, the IPs in the PL are AXI-4 and the Zynq is AXI-3, so for connection, the AXI interconnect is needed.

3.3.8 AXI SMC

The AXI smart connect Core is used to connect only AXI memory-mapped devices together.

3.4 DataMover PS program

For checking the data movers, a simple BareMetal application is developed in C language. The BareMetal not only provides less overhead than operating system platforms, but offers more deterministic behavior than OS application since there is no context switching.

The BareMetal application for the data mover is developed in the Xilinx SDK. The exported hardware from the Vivado generated bitstream is imported to Xilinx SDK, forming the base configuration for the project application.

For the data mover to transfer the data to, there is a need to define memory region at the DDR and OCM memory. For the former with the use of `unsigned long long ddr[DATA_COUNT];` the variable is defined and for the latter, `unsigned long long * OCM = (unsigned long long *) 0xFFFF0000` the OCM memory is allocated.

In the main body of the code, the function myinit_dma is called to instantiate the DMA and setting the DMA notification mode to polling and disabling the interrupt signals.

```
int myinit_dma(XAxiDma* AxiDma,u32 DMA_ADDRESS)
{
	XAxiDma_Config *CfgPtr;

	int status;

	CfgPtr = XAxiDma_LookupConfig(DMA_ADDRESS);
	if(!CfgPtr) {
			print("erro in configuring the DMA %x \n",DMA_ADDRESS);
			return XST_FAILURE;
```

```
        }

        status = XAxiDma_CfgInitialize(AxiDma,CfgPtr);
        if(status != XST_SUCCESS){
                print("Error initializing the DMA %x \n",DMA_ADDRESS);
                return XST_FAILURE;
        }

        XAxiDma_IntrDisable(AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
        XAxiDma_IntrDisable(AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);

        return XST_SUCCESS;
}
```

Next, the function init_counter is called. It instantiates the HLS counter, to have an incrementing counter it sets the operand to 1 then configures the start value and finally puts the bust size to dataLen.

```
int init_counter(XCounter* axiCounter,u16 counterAddr,u32 dataLen)
{
        XCounter_Config *CfgPtr;

        int status;

        CfgPtr = XCounter_LookupConfig(counterAddr);
        if(!CfgPtr) {
                print("Error looking for AXI counter config\n");
                return XST_FAILURE;
        }

        status = XCounter_CfgInitialize(axiCounter,CfgPtr);
        if(status != XST_SUCCESS){
                print("Error initializing counter\n");
                return XST_FAILURE;
        }
        XCounter_Set_operand_V(axiCounter,1);
        XCounter_Set_start_value_V(axiCounter,0x0);
        XCounter_Set_LEN_V(axiCounter,dataLen);
        return XST_SUCCESS;
}
```

Next, the code instantiates the GPIO and configures the direction as output and finally asserts the output to enable the counter.

```
status = XGpio_Initialize(&Gpio, XPAR_GPIO_0_DEVICE_ID);

if (status != XST_SUCCESS) {
                xil_printf("Gpio Initialization Failed\r\n");
                return XST_FAILURE;
}
XGpio_SetDataDirection(&Gpio, 1, ~1);
XGpio_DiscreteWrite(&Gpio, 1, 1);
```

After GPIO setting, the code initializes and enables the AXI timer with the following piece of code.

```
status = XTmrCtr_Initialize(&AxiTimer, XPAR_AXI_TIMER_0_DEVICE_ID);
XTmrCtr_SetOptions(&AxiTimer, XPAR_AXI_TIMER_0_DEVICE_ID, XTC_ENABLE_ALL_OPTION);
```

```
if(status != XST_SUCCESS){
        print("Error: timer setup failed\n");
}
```

By calling the function Xil_SetTlbAttributes(OCM,0x10c06); the memory attribute for the desired memory region can be modified and in this region, memory invalidations is not needed.

After setting all the parameters, before initializing the DMA transfer, the timer is reset and its values is read. Resetting and reading the timer is accomplished via the following piece of code:

```
XTmrCtr_Reset(&AxiTimer, XPAR_AXI_TIMER_0_DEVICE_ID);
time_A = XTmrCtr_GetValue(&AxiTimer, XPAR_AXI_TIMER_0_DEVICE_ID);
```

The DMA is instructed to transmit the desired amount of data from device to memory. After commanding the DMA, since the DMA mode is set to polling, in a while loop the program waits for the DMA to completes its transfer. In parallel with the DMA transfer, the timer is counting the duration of time based on the FCLK, by which the transfer rate with simple calculation can be measured. The DMA transfer and polling can be done by the following piece of code:

```
status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned  long long)ddr, dma_size, XAXIDMA_DEVICE_TO_DMA);

        if (status != XST_SUCCESS) {
                xil_printf("Error: DMA transfer from accelerator failed %d\n",status);
                return XST_FAILURE;
        }
        while (XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) ;
```

After the DMA transfer completes, to measure the DMA performance, the timer is read again and the second value is subtracted from the first timer readings. The measured value in terms of cycle-count is printed. The second timer reading and printing is done by the following piece of code:

```
time_B = XTmrCtr_GetValue(&AxiTimer, XPAR_AXI_TIMER_0_DEVICE_ID);
xil_printf("Runtime  is %d cycles.\n", time_B-time_A);
```

3.5 DataMover Transfer-rate

With the counter IP test design, ACP and HP port are tested separately. Both ports are configured at 64-bit width mode and tested with three burst size 23 KB, 64KB, and 67 MB running at 100MHZ targeting OCM and DDR memory. Memories are tested with different memory attributes, transferring data with invalidation, without invalidation and finally non-cacheable memory. As shown in the Figure 27 and Figure 28, in general both port performance is similar. Both ports targeting OCM and DDR memory locations with cache invalidation without considering the burst size, can have roughly bandwidth of 300MB/s. On the other hand, simple transfers and non-cacheable memory transfers can have a bandwidth up to 400MB/s. Simple transfers are invisible to CPU and these transfers are not practical in this project.

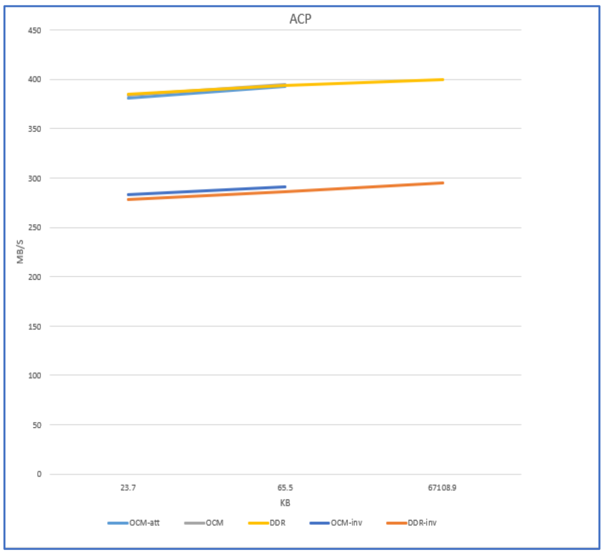


Figure 28-ACP port measurement

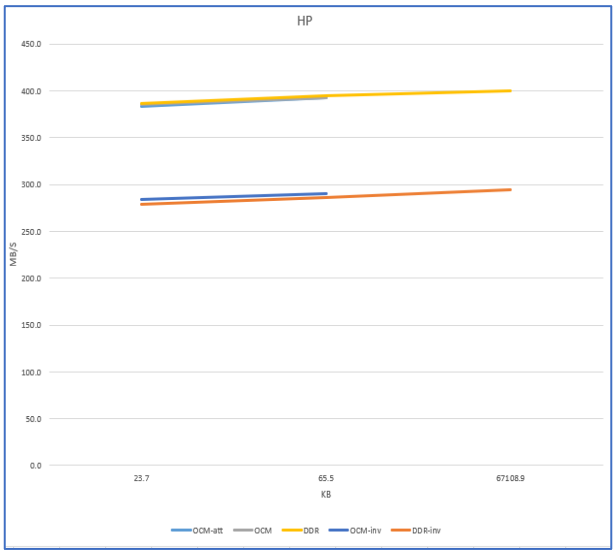


Figure 27-HP port measurement

In this project, a conclusion can be made that, any data mover with cache invalidation or with memory attribute configuration is a good choice, because any one of the mentioned methods has a performance almost three to four times of the Gigabit ethernet.

3.6 ZynqBoard PS program

After moving data from PL to PS with the best date mover, the data is needed to be encapsulated as TCP socket[3] format and be sent via ethernet link. There are at least three feasible methods for achieving this goal:

1. Petalinux:

   Petalinux is the Xilinx Linux distribution targeted for Zynq series and Microblaze CPUs. Where the Linux comes in to the picture, it gives the easy feeling of providing already known POSIX environment to deal with. On the other hand, it gives the complexity to write drivers in the Linux kernel. [11]

   Linux kernel is monolithic, it has two memory regions, kernel space and user space. The application runs in the user space and if it needs to access the hardware, it needs to go through kernel by the system calls or APIs. In this project using DMA is inevitable. With using the Memory map function the DMA directly is accessible from the user but, there is a need for consistent memory, accessible by the application.

   The application address space in Linux is virtual and the memory the DMA has access through using ACP or HP port is physical. Having said that, Linux OS with two distinct memory regions cause the DMA driver to do additional memory copy, which adds cost to overall performance. [12]

3 The term socket is referred to a representation of communication link in terms of Unix file descriptor Standard.

2. BareMetal LWIP RAW mode:

   LWIP is an open source TCP/IP library specially designed for embedded systems and it can run in two modes, RAW mode which is a none-thread environment in super loop and socket mode that can easily run in multithreaded environment. [13] Good news is the Xilinx SDK supports both of these modes.

   Xilinx has conducted thorough case study on running LWIP with different modes on their different hardware designs. [14] As it can be seen in Table 12, the last column shows the LWIP performance in the Zynq series hardware platform. With the first glance it can be concluded that RAW mode with this performance result, which is almost double than the socket mode, is better. But these numbers are achieved with full CPU usage.

Table 12-LWIP TCP performance [14]

Hardware Design Name	**RAW Mode**		**Socket Mode**	
	RX (Mb/s)	**TX (Mb/s)**	**RX (Mb/s)**	**TX (Mb/s)**
AC701_AxiEth_32kb_Cache	205	125	37	41
AC701_AxiEth_64kb_Cache	270	175	40	46.8
KC705_AxiEth_32kb_Cache	290	190	52.9	56.2
KC705_AxiEth_64kb_Cache	380	250	58.4	69.5
KC705_AxiEthernetlite_64kb_Cache	46	67	29	44
ZC702_GigE	943	949	521	542

3. BareMetal LWIP Socket mode:

   For using and testing the Socket mode, a multithreaded environment is required and to have this platform, FreeRTOS 10 is used. It has a Berkeley socket API. This API is very user friendly and any POSIX programmer is familiar with. Moreover, a multithreaded environment gives a more deterministic behavior platform.

For this project due to the complexities of Linux kernel development and shortage in time, exploring the petalinux is out of option and BareMetal LWIP is preferred. Based on the Xilinx performance results, as shown previously, it can be concluded that the main performance bottleneck in this project is the application region and it must be dealt with care.

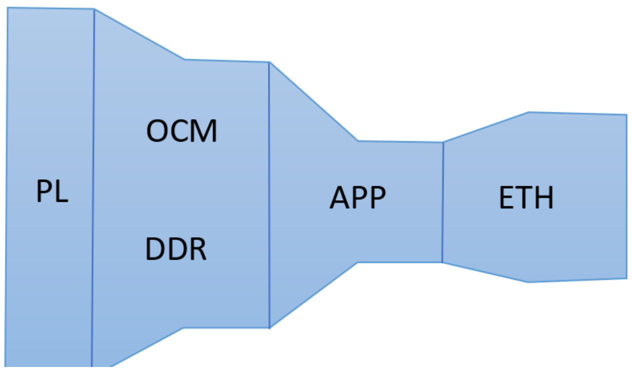


Figure 29-performance bottleneck diagram

For implementing the program in both LWIP raw and socket mode, the corresponding Xilinx template projects for each type are used. The projects templates offer the TCP iperf testing platform by which the network link throughput can be measured. [15]

In this project, for evaluating the ZynqBoard Gigabit Ethernet link, the iperf test as a TCP client in both RAW and SOCKET LWIP is performed and the same Xilinx measurements shown in Table 12 were witnessed.

As shown in Table 12, the LWIP library in RAW mode can have a speed almost up to full bandwidth and seems to be the perfect choice for this project.

Unfortunately, after combining the data mover part and this mode, the ethernet throughput subsided in a way that the bandwidth speed is dropped below 100Mbps. This situation was explained previously by Xilinx which was mentioned above. This unexpected drop in bandwidth is due to LWIP using 100% CPU and due to any modification in the super loop, leads to overhead in this mode.

For better clarification of the raw mode sensitivity, the following simple for loop is added to the transmission super loop and the transmission rate is dropped from 930Mbps to lower than 75Mbps.

```
for (i = 0; i < TCP_SEND_BUFSIZE; i++)
      send_buf[i] =(i % 10) + '0';
```

With this outcome the conclusion can be made that, working with LWIP RAW mode needs to be dealt with care and developing an ethernet application with desirable throughput needs a lot of effort, which is beyond the scope of this project.

The other mode of the LWIP is socket mode. This mode is a little bit more complex than the RAW mode since, it runs in the multithreaded environment. LWIP Socket mode API is Berkeley socket-like and with a handful simple functions calls the desired functionality, regardless of the transmission protocol, is reached. [16]

In this project, although the throughput of the socket mode is half of the Gigabit ethernet media, it is sufficient enough for the desired functionality. [14]

To use LWIP in socket mode, same as the RAW mode Xilinx template sample design is used and altered to reach the expected usage. In socket mode, the main body of the LWIP is handled in an individual thread which is spawned at the beginning of the program. Later on, the configuration and packet transmission are done and called via other threads.

The GTX burst size is set to 26*69 and the IP is enabled via the following function calls

```
GTX_mWriteReg(GTX_ADDRESS,GTX_S_AXI_SLV_REG0_OFFSET,1794);
GTX_mWriteReg(GTX_ADDRESS,GTX_S_AXI_SLV_REG1_OFFSET,0x12340000);
```

In this implementation, as shown in the following excerpt code, it is intended to use all four OCM memories on the high address space in which DMA transfers to. Each transfer is commenced sequentially in non-blocking mode. The sequential order is important to keep the data integrity.

```
Xil_Out32(XSLCR_UNLOCK_ADDR, XSLCR_UNLOCK_CODE);
```

```
Xil_Out32(XSLCR_OCM_CFG_ADDR, 0x1F);
Xil_Out32(XSLCR_LOCK_ADDR, XSLCR_LOCK_CODE);
Xil_SetTlbAttributes(0xfff00000,0x10c06);
```

For configuration, in order for the OCM to have access to high address space, the SLCR register must be unlocked, the OCM_CFG must be set and after configuration the SLCR is locked again preventing any further modification.

After commencing DMA transfer, the status of the corresponding DMA transfer is checked, if it has been completed, the ready signal for that region is asserted notifying the TCP transfer to use this region. TCP transfer after using any region, will notify the DMA that it has successfully transferred the data. The DMA will commence another transfer from PL to PS to use the region and the cycle continues.

In this project, DMA transfers and TCP transfers are implemented in one thread to obtain a better performance. Using two threads or more adds unavoidable synchronization and contention overheads to the system which leads to performance degradation.

The lwip_write function is the main part of the LWIP library which transfers the data on the socket. One of the arguments of this function is the length of the payload. It has been witnessed that optimal payload size is better to be around five times of the TCP maximum segment size (5*1460 Byte) and this payload size has been used and implemented by Xilinx in LWIP templated designs. Each ADC board payload size is 69 words (69*4 Byte) and to omit ADC board payload fragmentation in the TCP transfers, the transfer length is set to 7176 Byte (69*4*26) which is close to optimal transfer rate.

In general, any TCP transfer is prone to failure, due to network congestion, recipient buffer occupancy and many more issues. The function return value specifies the successful byte transfers and by keeping track of each lwip_write all packets can be transferred.

```
while (1) {
    if (commit[dma_loop] == 0 && ready[dma_loop] == 0) {
            status = XAxiDma_SimpleTransfer(&AxiDma, send_buf_rec[dma_loop],
                            max, XAXIDMA_DEVICE_TO_DMA);
            if (status != XST_SUCCESS) {
                    xil_printf("Error: DMA num %x transfer from SFP failed %x\n",
                                    dma_loop, status);
            } else {
                    commit[dma_loop] = 1;

            }
    }
    if (commit[dma_loop] == 1) {
            if (!XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) {
                    commit[dma_loop] = 0;
                    ready[dma_loop] = 1;
                    if (dma_loop < buffer_count - 1)
                            dma_loop++;
                    else if (dma_loop == buffer_count - 1)
                            dma_loop = 0;
            }

    }
    if (ready[tcp_loop] == 1) {
            memcpy(payload, send_buf_rec[dma_loop], max);

            while (total_sent < data_lenght) {
                    if ((bytes_send = lwip_write(sock, payload + total_sent,
                                    data_lenght - total_sent)) < 0) {
                            xil_printf("TCP Client: Either connection aborted"
                                            " from remote or Error on tcp_write\r\n");
```

```
                                u64_t now = sys_now() * portTICK_RATE_MS;
                                u64_t diff_ms = now - client.start_time;
                                tcp_conn_report(diff_ms, TCP_ABORTED_REMOTE);
                                break;
                        }
                        total_sent += bytes_send;
                }
                total_sent = 0;
                ready[tcp_loop] = 0;
                if (tcp_loop < buffer_count - 1)
                        tcp_loop++;
                else if (tcp_loop == buffer_count - 1)
                        tcp_loop = 0;
        }
    }
```

Unfortunately, it has been witnessed that when the memory region attribute is set to 0x10c06, the LWIP cannot use this memory region. For resolving this issue, another transfer from OCM to DDR memory is required which adds a cost to overall performance. LWIP has another issue. It has an internal buffer and locates it in the DDR memory, in another words, by each transfer, it will copy the specified data to buffer, which by nature is another drawback to this project. As shown in Figure 30, complete visual memory data flow can be seen.

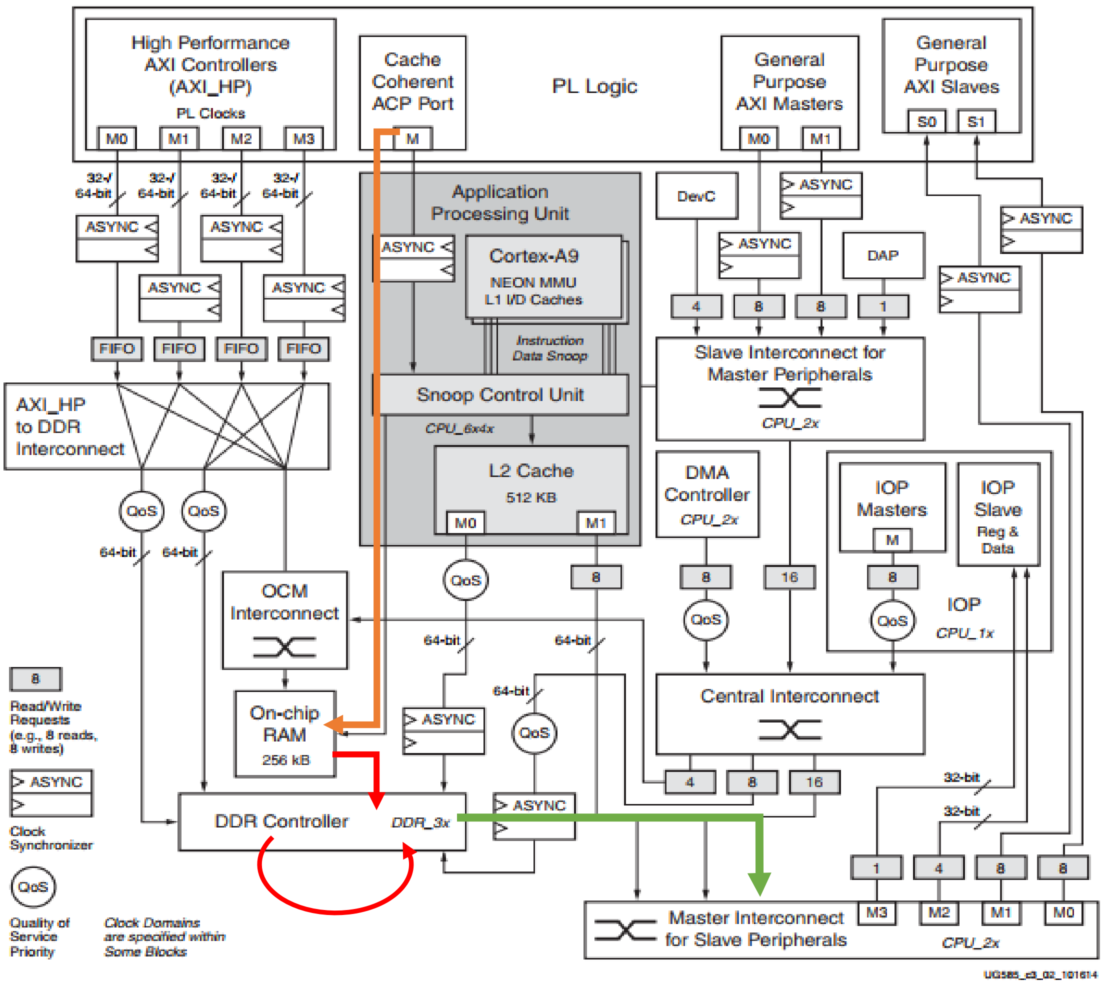


Figure 30-Data Flow

3.7 Linux Host program

In the final part, a simple server application on the Linux host is implemented to receive the data and store it. The application is written in C language, which utilizes the Linux socket API with file system calls to write in raw binary format on the local storage. Since storing this humongous amount of records on the RDMBS is cumbersome, the raw binary format is chosen.

As demonstrated in Figure 31, in the Linux kernel, or better saying any operating system, there is a specific TCP RX TX buffer specific to each TCP connection and in Linux the size of the buffers can be easily modified. When the server application bonds and listens on the specific port, it enables the appropriate buffer for each stablished connection. When the ZynqBoard client is connected to server, the packets are sent to server TCP RX buffer and wait for the server application to fetch them. If the buffer is full it will block the Zynboard transfer. This blocking mechanism in the TCP transmission enables complete flow control in every stage back to the sender which here is the ADC board. Blocking mode makes the link steady and reliable and guarantees every bit to be transferred and finally stored.

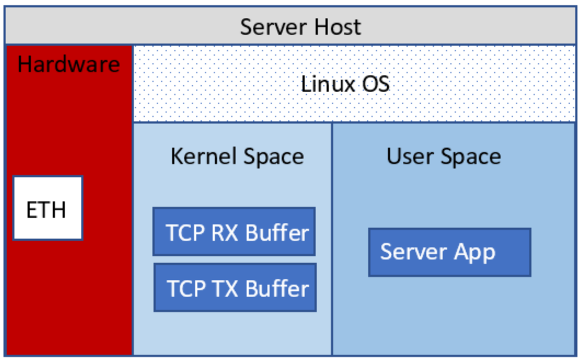


Figure 31-Linux TCP overview

The program listens to TCP port number 5001 and bonds on any available interface on the Linux host. After initial settings and configurations, the program gets ready to receive. After each fetch, the program will write the data to file in the binary format. Since the byte order of Intel architecture is little-endian and, in this project, the same configurations are applied to the ARM cores, there is no need to change the bytes to network order and simply after receiving, data can be stored. The recv function call is not successful on every call. Fortunately, the function returns the number of fetched bytes and to make the receive mechanism functional, it must be called in the loop to fetch as much as expected by monitoring the bytes. [17]

```
while(1){
   for (;total_rec<size;) {
                        my_int=0;
                        ok=recv(sockfd,x+total_rec,size-total_rec,0);
                        if(ok<0)
                                 break;
                        total_rec+=ok;
             }
             fwrite(x, max, 1, write_f);
             total_rec=0;
}
```

4 Test and debugging

To check the correctness of any design, based on the description and specifications it needs to be tested. In this project, which is a comprisal of hardware and software, each part needs to be verified and finally, the complete system must be evaluated.

In addition, in hardware and software development debugging tools are the necessity of the design. In this section, tests and some available tools for debugging are described.

4.1 BERT

Every high-speed serial design demands specific testing and procedures such as data rate, Bit error rate, Jitter and Eye diagram. [18]

The test is conducted in pre-design and post-design stages. In the former the tests are done through IBIS-AMI simulations model. With IBIS each high-speed serial transceiver manufacturer provides its product model to the costumers without revealing their IP design and the designer can run the design on any third-party software capable of simulation. Xilinx provides its 7 series transceiver IBIS-AMI model but since in this project the ZynqBoard is already designed, performing simulation is unjustifiable.

After the design, there are post-design tests and verification which must be performed based on the design application. In most designs it is required to not only test the internal functionality of the transceiver, but also to test the whole system design from transceiver to PCB traces, connectors (SMA, SFP, BNC, …), and cables. [19]

One post-test is Bit Error Rate Test (BERT) which is based on the Bit Error Rate (BER). BER is the ratio measuring the bit errors occurring in the specific number of bit transmissions, which usually should be equal or less than 10^{-12} . In general, the mechanism would be such that at the source pseudo-random binary sequence is generated and fed to the transceiver and at the receiver the expected sequence is evaluated and the ratio is measured. On the other test, Eye Diagram or Eye Pattern is the display of the received digital differential signal. Displaying an Eye diagram on a simple oscilloscope can be achieved by manipulating the sweep in a way that both signal overlaps. In advanced oscilloscope Eye diagram is an addon feature. Via the Eye diagram observation, the quality of the link in terms of jitter, quality factor, signal to noise and many more can be analyzed.

Since the high-speed serial transceivers, as explained in the previous section, can run up to 28 Gbps, the clock frequency of the transceiver is a bit higher than the bit rate. Testing the transceivers with such high clock frequencies, requires high-speed test equipment that in some application is not feasible. Good news is, high-speed transceiver vendors such as Xilinx, provide a build-in mechanism in their transceiver that alongside with their IDE, tests like BERT can be performed and with extrapolation from the BERT the eye pattern can be generated.

In 7 series Xilinx transceivers as shown in Figure 32, in the PCS section in parallel to the receiver data path without any interruption to receiver, reads the error rate from equalization section of PMA by which it can form sample and error counter. With this mechanism the BER ratio is measured and with extrapolation from BER, phase offset and voltage offset, the horizontal and the vertical axis of the eye pattern is formed and the diagram can finally be generated.

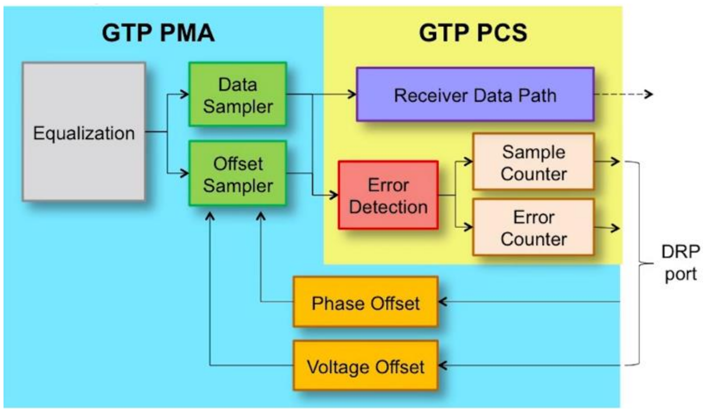
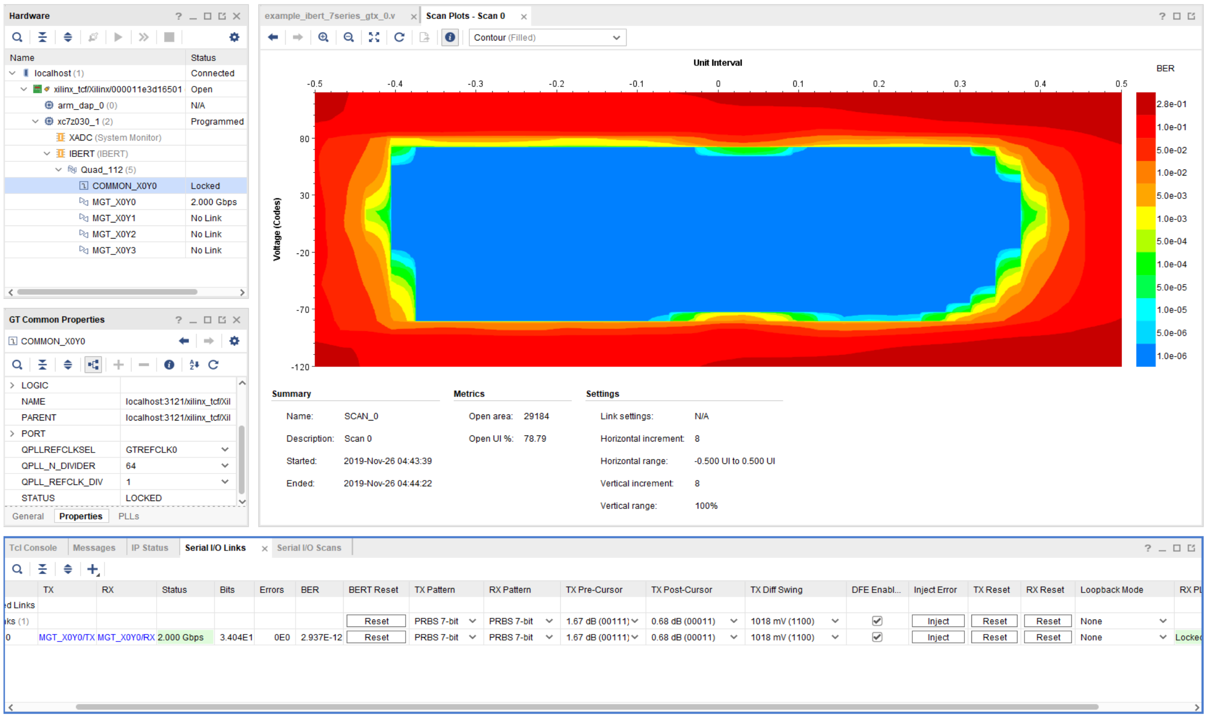


Figure 32- 7 series eye pattern generation [23]

Vivado has a specific "IBERT 7 series GTX "IP in its catalog repository by which Integrated bit error ratio tester can be instantiated and based on the desired speed and protocol can be configured. [20]

After instantiation and programming the FPGA as it can be seen in Figure 33, the IBERT can be accessed via Vivado program hardware manager through JTAG link. It has a wizard by which the user can configure and build up the links and manipulate the PLL settings, inject error, reset RX TX region, feedback control, draw eye scan and many more.



Name	Status
localhost (1)	Connected
xilinx_tcf/Xilinx/000011e3d16501	Open
arm_dap_0 (0)	N/A
xc7z030_1 (2)	Programmed
XADC (System Monitor)	
IBERT (IBERT)	
Quad_112 (5)	
COMMON_X0Y0	Locked
MGT_X0Y0	2.000 Gbps
MGT_X0Y1	No Link
MGT_X0Y2	No Link
MGT_X0Y3	No Link

LOGIC	
NAME	localhost:3121/xilinx_tcf/Xil
PARENT	localhost:3121/xilinx_tcf/Xil
PORT	
QPLLREFCLKSEL	GTREFCLK0
QPLL_N_DIVIDER	64
QPLL_REFCLK_DIV	1
STATUS	LOCKED

Summary

Name: SCAN_0
Description: Scan 0
Started: 2019-Nov-26 04:43:39
Ended: 2019-Nov-26 04:44:22

Metrics

Open area: 29184
Open UI %: 78.79

Settings

Link settings: N/A
Horizontal increment: 8
Horizontal range: -0.500 UI to 0.500 UI
Vertical increment: 8
Vertical range: 100%

	TX	RX	Status	Bits	Errors	BER	BERT Reset	TX Pattern	RX Pattern	TX Pre-Cursor	TX Post-Cursor	TX Diff Swing	DFE Enabl...	Inject Error	TX Reset	RX Reset	Loopback Mode	RX Pl
d Links																		
ks (1)							Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	1018 mV (1100)	✓	Inject	Reset	Reset	None	
0	MGT_X0Y0/TX	MGT_X0Y0/RX	2.000 Gbps	3.404E1	0E0	2.937E-12	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	1018 mV (1100)	✓	Inject	Reset	Reset	None	Locke

Figure 33-Vivado 7 series IBERT wizard

In this Project as it can be seen in the following eye diagrams, the ZynqBoard IBERT is tested at 2Gbps, 4Gbps, and 6.25Gbps speed rates.[4]

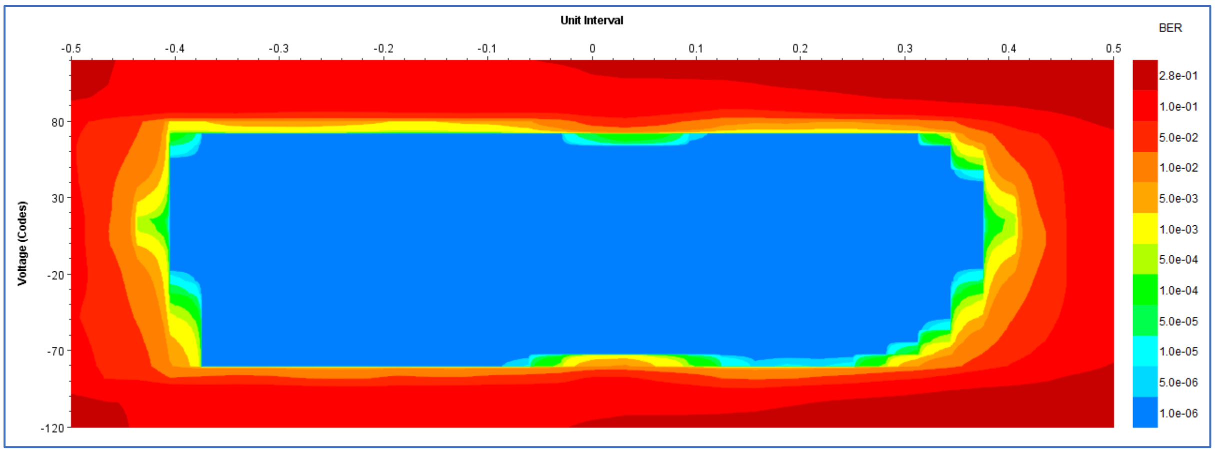


Figure 34-2Gbps Eye diagram

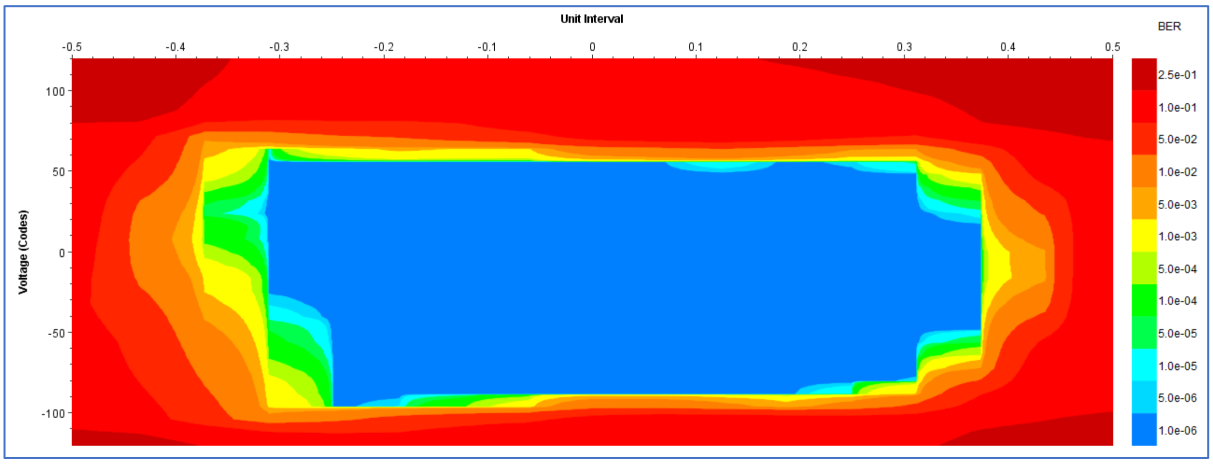


Figure 35-4Gbps Eye diagram

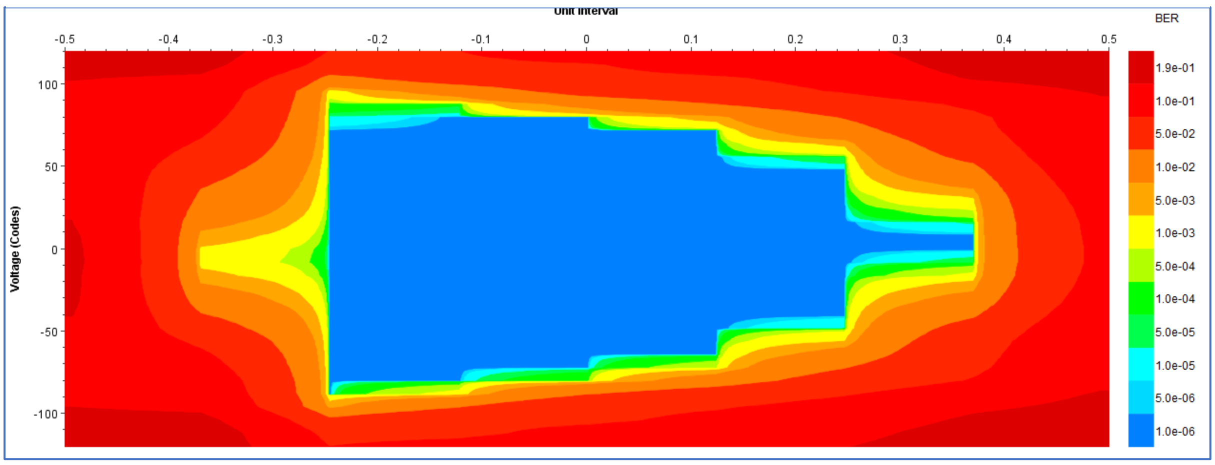


Figure 36-6.25Gbps Eye diagram

[4] The IBERT projects are designed by Pawel Marciniewski

4.2 ILA

In digital designs, to perform debugging or testing, it is important to know about the status of the signals and the buses. To check the external digital signals and buses, device called logic analyzer is used but what if there is a need to check the internal signals? The FPGA vendors for accomplishing this have devised a mechanism to check the internal signals. Xilinx has an IP called Integrated Logic Analyzer (ILA) for monitoring the internal signals based on simple and advanced trigger system. Simple trigger mode will trigger the ILA to capture signals based on bitwise operations like AND, OR, XOR and NOR or equality or non-equality to a constant number or rising and falling edge. On the other hand, the advanced trigger will control the ILA trigger based on the finite state machine up to 16 steps. In this mode, ILA also utilizes four 16-bit counter and four flags combined with up to three-way conditions.

In this project, ILA had a significant role for debugging. To check the data integrity of the streams of the data in each step of the logic, the advanced trigger, as shown in Figure 37, has been used. In this state machine, based on the selected probe which here is the high-speed transceiver, the received data have been checked for the expected ADC data pattern that, after the header, if the 3XXXXXXX word doesn’t show up the ILA will be triggered. [21]

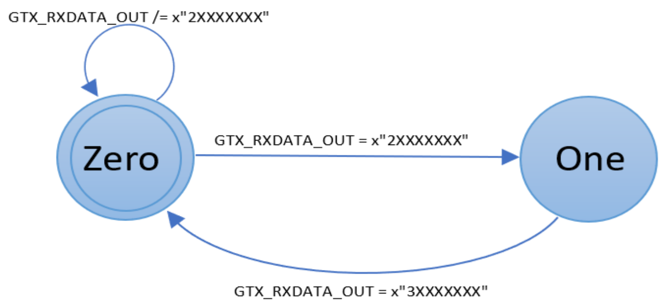

GTX_RXDATA_OUT /= x"2XXXXXXX"
Zero
GTX_RXDATA_OUT = x"2XXXXXXX"
One
GTX_RXDATA_OUT = x"3XXXXXXX"

Figure 37-ILA trigger FSM

To check the snapshot of the received data or the FIFO state machine, the simple trigger mechanism has been used.

4.3 VIO

To feed data to logic, usually buttons and dip switches are used and for displaying the system output LEDs, seven segments and LCDs are utilized. But, sometimes, there are no such devices available on the production hardware like the ZynqBoard. To overcome this issue, Xilinx introduces Virtual Input Outputs (VIO) IP. With VIO simple inputs and outputs with different width can be established. Even single bits inputs can be either identified as active high or low buttons.

In this project, for debugging high-speed flow control and bandwidth monitoring, VIO has been used. Using VIO saves a lot of debugging time since modifications will be dynamic. [21]

4.4 ChipScope Pro Analyzer

ChipScope Pro Analyzer is another Xilinx debugging tool which was previously deployed with ISE environment. ChipScope with its logic core, will build up a connection to FPGA and capture the desired signals.

In comparison to Vivado debugging environment, ChipScope is very fast, it can communicate with all FPGAs sitting on the JTAG bus, and it doesn't glitch when all system analyzer, virtual I/O low-profile software cores and logic analyzers are working together.

Using ChipScope in Vivado is a bit tricky, since the cores must be generated in ISE environment and instantiated in Vivado.

The ADC board firmware was previously developed by ChipScope cores to facilitate logic analyzing and BERT. In this project, the ChipScope not only made the usage of ADC board possible, but also provided thorough monitoring system for this board.

As shown in Figure 38, with the help of ChipScope it was possible to monitor if ZynqBoard is sending the busy signal to the ADC board and what data is being sent by ADC board on any high-speed links.

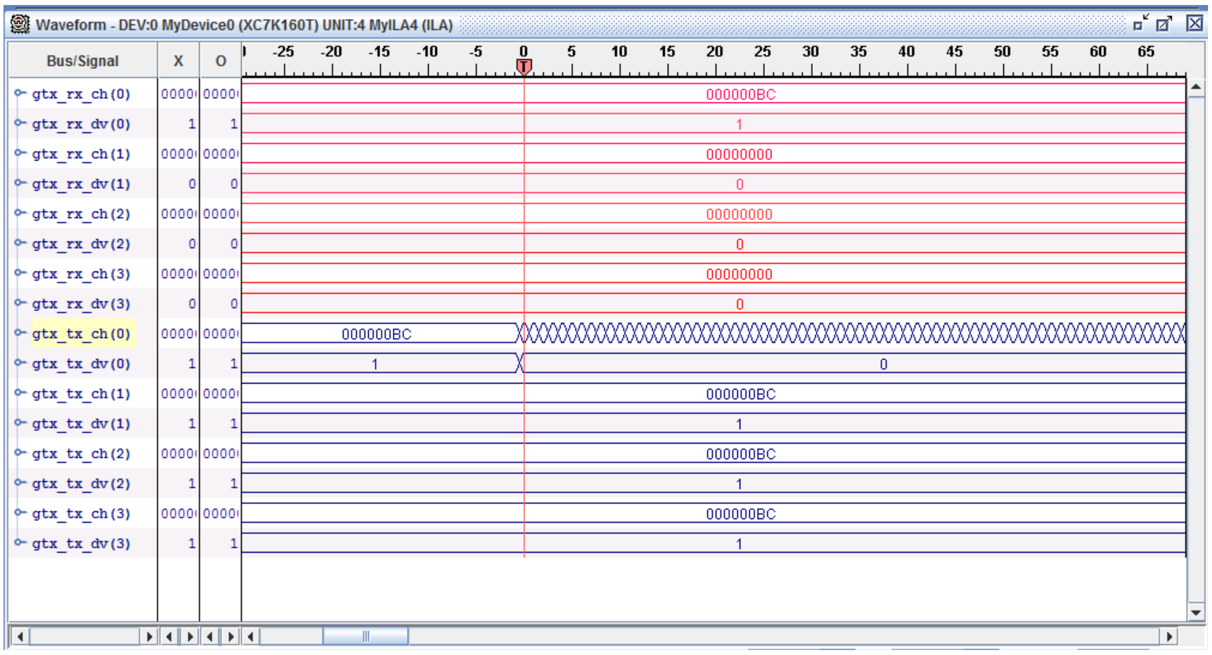


Figure 38-Chip Scope monitoring High-speed Transceiver

4.5 System integrity test

In this project, there is no margin to lose a single bit of data in transmission. To accomplish such a system, each part must be performed in a blocking manner; from the recipient host storing data to PL and to the sender ADC board. To check data integrity at this level a simple test can be performed. Instead of the ADC board sending the data, the counter IP that was used in the data mover part, is used to make the consecutive data pattern and the receive host program expects the pattern and checks the correctness of the whole data flow. After sending 100GB of data, not a single error was witnessed.

```
for(i=0;i<max;i+=8){

    unsigned long long *current=(unsigned long long *)(x+i);
      if(temp!=*current-1)
                                          {
                                            temp=*current;
                                                    printf("loop:%llu\r\n",*current);
                                                    print_c++;
      }
}
```

5 Performance

The data are being received and stored in the expected format and to monitor their throughput at the Linux host, system monitor tool has been used. As it can be seen in Figure 39, the network histogram shows that the data are being received with the average throughput of 60 MB/s which is almost the maximum data throughput achievable by LWIP socket mode.

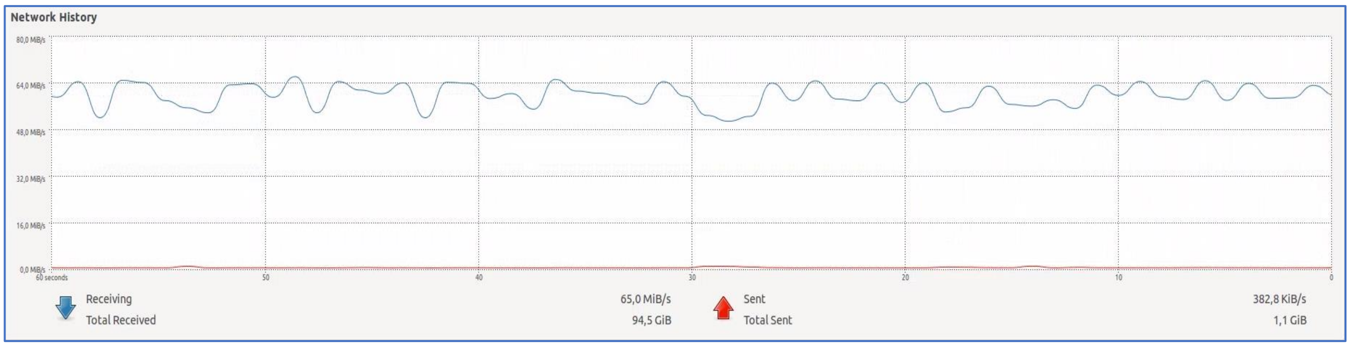


Figure 39- Linux Host network history diagram

6 Future work

The following improvements in software and hardware can be introduced:

Software:

- OpenAMP or metal with the LWIP:

  Zynq architecture supports asynchronous mode by which any core can run independently. To facilitate from this feature in this project, asynchronous computing libraries like OpenAMP or metal with the LWIP raw mode can be used.

  By offloading data movement process to another core, it will enhance the performance of Gigabit Ethernet.

- Linux

  Customized Linux can be used as a platform in this project. Linux kernel utilizes both cores with optimal performance. Having SMP platform with rich developing libraries like pthread not only offers a thorough POSIX platform but also provides an environment that a lot of third-party software and libraries can be used.

- Accelerating the LWIP TCP section:

  The CPU centric part of the LWIP is the TCP section which can be offloaded to PL and be accelerated. With this mechanism the LWIP raw mode can perform better.

Hardware:

- Additional Ethernet port:

  Since the Zynq SOC supports two Gigabit Ethernet port and this version of the ZynqBoard is just using one Ethernet port, another port can be added to the next hardware design. Having two Gigabit port can used individually or in aggregation mode. In the aggregation mode, both ports are bonded together and forms 2Gbps interface. This feature can be implemented in the modern operating system like Linux.

- Changing the FPGA Chip

  Instead of using SOC family, a regular FPGA chip with more high-speed transceivers can be used. With this modification not only, the complication of the Zynq SOC is omitted but also a new high-speed interface link like USB 3.X, 10Gbit Ethernet or PCIe 3 or 4 can be introduced that via this interface the board communicates with the recipient host.

7 References

[1] "GSI Helmholtz Center for Heavy Ion Research," [Online]. Available: https://panda.gsi.de/article/electromagnetic-calorimetry.

[2] Xilinx(DS190), *Zynq-7000 SoC Data Sheet: Overview,* Xilinx, 2018.

[3] Xilinx(UG585), *Zynq-7000 SoC Technical Reference Manual,* Xilinx, 2018.

[4] Xilinx(UG1037), *Vivado Design suite Vivado AXI Reference,* Xilinx, 2017.

[5] ARM(AMBA®AXI), *AMBA® AXI™ and ACE™ ProtocolSpecification,* ARM, 2013.

[6] Xilinx, *DS191- Zynq-7000 SoC DC and AC Switching Characteristics,* Xilinx, 2018.

[7] Xilinx(UG476), *7 Series FPGAs GTX/GTH Transceivers,* Xilinx, 2018.

[8] Xilinx(UG902), Xilinx, 2016.

[9] Xilinx(PG021), *AXI DMA v7.1,* Xilinx, 2019.

[10] Xilinx(PG079), *AXI Timer v2.0,* Xilinx, 2016.

[11] UG1144, *PetaLinux Tools,* Xilinx, 2019.

[12] J. Madieu, Linux Device Drivers Development, Packt, 2017.

[13] A. Dunkels, "nongnu," 2018. [Online]. Available: https://savannah.nongnu.org/projects/lwip/.

[14] S. M. a. U. C. Anirudha Sarangi, *LightWeight IP Application Examples,* Xilinx, 2014.

[15] Jon Dugan, Seth Elliott, Bruce A...., "iperf.fr," [Online]. Available: https://iperf.fr/.

[16] Adam Dunkels, Leon Woestenberg, "https://www.nongnu.org/lwip/," [Online]. Available: https://www.nongnu.org/lwip/2_0_x/group__socket.html.

[17] B. Hall, Beej's Guide to Network Programming, Jorgensen Publishing, 2011 .

[18] L. E. F. Jr, Handbook of Serial Communication interfaces, Elsevier, 2016.

[19] H. F. a. R. Mayder, *Signal Integrity Simulation and On-Chip Evaluation for Low-Cost FPGA Transceivers,* Xilinx, 2017.

[20] Xilinx(PG132), *Integrated Bit Error Ratio Tester 7 Series GTX Transceivers v3.0,* Xilinx, 2016.

[21] Xilinx(UG936), *Vivado Design Suite Tutorial Programming and Debugging,* Xilinx, 2016.

[22] Xilinx, "7-series-dedicated-hardware," 2014. [Online]. Available: https://www.xilinx.com/video/fpga/7-series-dedicated-hardware.html.

[23] Xilinx, "Xilinx.com," 2015. [Online]. Available: https://www.xilinx.com/video/fpga/artix-7-transceiver-in-low-end-

device.html#:~:targetText=The%20FPGA%20industry's%20only%20low,design%20for%20cost %2Dsensitive%20applications..

[24] Xilinx(UG900), *Vivado Design Suite User Guide Logic Simulation,* Xilinx, 2018.