CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Microprocessor Communication Over Ethernet

Mini project submitted for 520l: System on Chip Design

By
Kambla Kethana Rao
201996398

Table of Contents

List of Figures

List of Tables

Abstract
Microprocessor Communication Over Ethernet

Microprocessors are fabricated on a small chip which is capable of performing Arithmetic Logical Unit (ALU) operations. The microprocessor is also capable of communicating with other devices that are connected to it. The first microprocessor Intel 4004 was introduced in 1971. VLSI circuit with loaded trillions of electronic components on a chip in size to large scale integration circuit was introduced 10 years ago.

The ZYBO (ZYnq Board) is a development board developed by Xilinx company. It belongs to their famous Xilinx Zynq-7000 called Z-7010, that has rich features and ready to use embedded software. ZYBO was developed in 2013. Few years later ZYBO was upgraded to next generation that is now called as ZYBO Z7. The Zybo Z7 development bord has multiple set of multimedia and connectivity peripherals. Zybo Z7 consists of 667 MHz dual-core Cortex-A9 processor. Zybo Z7 has various ports and connectivity. This digital development board supports Ethernet, micro USB, Audio and Video connectivity. This board takes helps of Realtek RTL8211E-VL PHY to implement 10/100/1000 ethernet port for the network communication.

USB and Ethernet ports provided in the board are used to communicate between the board and other device with the help of software suite called Xilinx Vivado, which was introduced in April 2012.

Hardware Description Language commonly known as HDL is used to describe the behavior structure and of digital electronic circuits. Vivado software can be used in synthesis and analyzation of HDL designs.

This project aims to implement capturing network packets over ethernet using Xilinx Vivado 2018.3 and Wireshark software

CHAPTER 1 INTRODUCTION

1.1 Zynq Development Board

The Zybo Z7 board is a digital circuit development board which has various built-in ports, connectors, and connectivity peripherals. Zybo Z7 development board has ARM Cortex-A9 processor. This dual- core ARM processor has a DDR3L memory controller. This consists of total 8 DMA channels and 4 AXI3 slave ports.

Zybo-Z7 board has a 1G ethernet port, USB, and as well as SDIO ports that makes up zybo to have a high bandwidth peripheral controllers. Whereas Low bandwidth peripheral controllers of Zybo consists of SPI, UART, CAN and I2C.

Zybo z7 has 1 GB DDR3L memory with 32-bit bus that operates at 1066 MHz and 16 MB Quad-SPI Flash memory. Zynq development board has USB-UART bridge that USB OTG PHY connection with the host and computer. We can see from the figure 1.1, it has 6 push buttons and 4 slide switches. It has 5 LEDs and 2 RGB LEDs. When it comes to powering the Zybo z7, we can use USB or any external sources that has 5V to power it on.

The detailed diagram of zybo z7 can be seen below in Fig 1.1-154

Figure 1.1 Diligent zybo z7 development board [01]

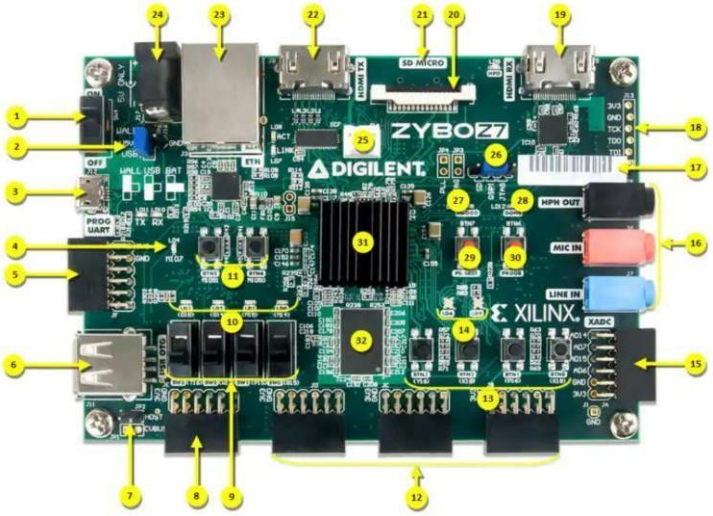


Table 1.1 Call out description of Zybo z7 board

Callout	Description	Callout	Description
1	Power Switch	17	Unique MAC address label
2	Power select jumper	18	External JTAG port
3	USB JTAG/UART port	19	HDMI input port
4	MIO User LED	20	Pcam MIPI CSI-2 port
5	MIO Pmod port	21	microSD connector (other side)
6	USB 2.0 Host/OTG port	22	HDMI output port
7	USB Host power enable jumper	23	Ethernet port
8	Standard Pmod port	24	External power supply connector
9	User switches	25	Fan connector (5V, three-wire) *
10	User LEDs	26	Programming mode select jumper
11	MIO User buttons	27	Power supply good LED
12	High-speed Pmod ports *	28	FPGA programming done LED
13	User buttons	29	Processor reset button
14	User RGB LEDs *	30	FPGA clear configuration button
15	XADC Pmod port	31	Zynq-7000
16	Audio codec ports	32	DDR3L Memory

1.2. ZYBO Z7-10

The Zybo Z7 can be seen in two

versions: 1. Zynq-7010

2. Zynq-7020

These two versions of Zybo board are usually termed as Zybo Z7-10 and Zybo Z7-20 respectively.They can be collectively referred as Zybo.

The size of FPGA inside Zync AP SoC forms the primary difference between the two variants. Z7-20 has about 3 times larger internal FPGA compared to its predecessor Zybo Z7-10. Also, the Z- 20 variant has more pins than the Z-10. Z-20 model has heat sink and a fan to remove excessive heat generated during the process which indicates the presence of several additional features on Zybo Z7- 20.

Figure 1.2 Zybo Z7-10 [02]

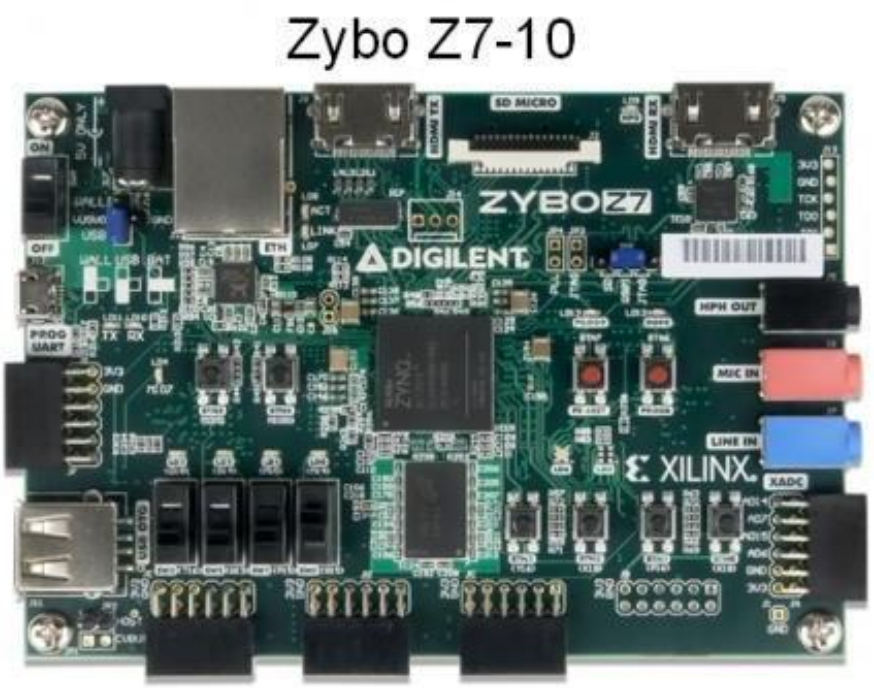


Table 1.2 Zybo z7-10 features

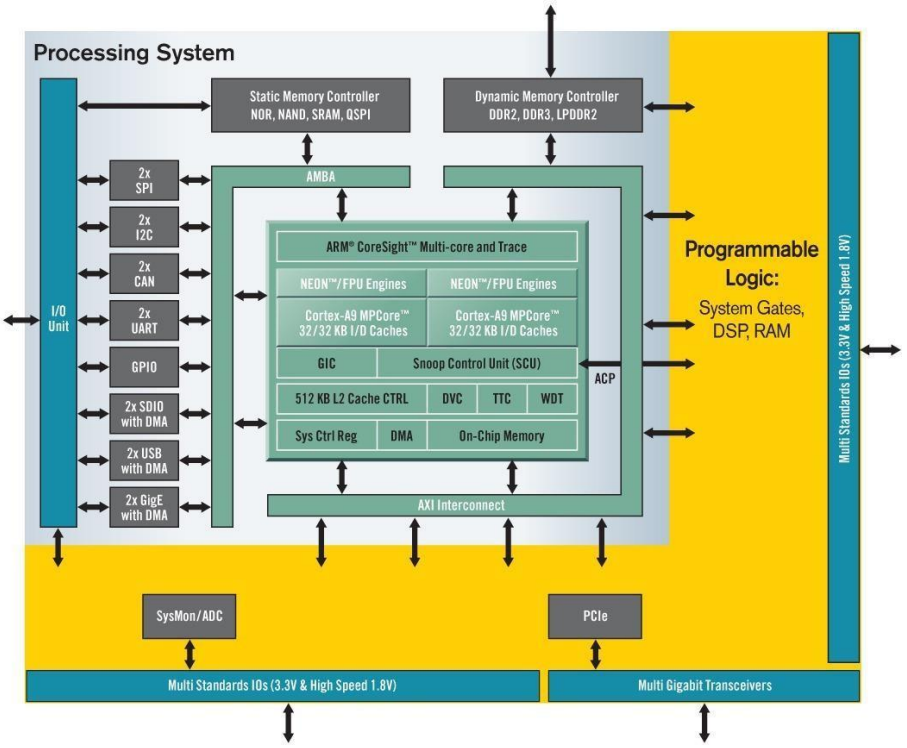
Product Variant	Zybo Z7-10
Zynq Part	XC7Z010-1CLG400C
1 MSPS On-chip ADC	Yes
Look-up Tables (LUTs)	17,600
Flip-Flops	35,200
Block RAM	270 KB
Clock Management Tiles	2
Total Pmod ports	5
Fan connector	No
Zynq heat sink	No

1.2 Zynq APSoC Architecture.

The Zynq APSoC consists of two distinct subsystems, The Processing System (PS) and The Programmable Logic (PL). The overview of Zynq APSoC architecture is shown below in Fig 1.3with Light green colored is PS and yellow is PL .

The PL contains several ports and buses dedicated to tightly couple it to the PS except for which it is nearly identical to Xilinx 7-series Artix FPGA with difference in hardware configuration. Either only processor can configure PL or JTAG port.

Figure 1.3 Zynq APSoC Architecture [01]



As you can see from the architecture, PS has Application Processing Unit (APU), Advanced Microcontroller Bus Architecture Interconnect, Memory Controller and other peripheral controllers with their inputs and outputs multiplexed to 54 dedicated pins (MIO pins). The unconnected peripheral controllers to MIO pins can be routed to their I/O via the Extended-MIO(EMIO) interface through the PL. The Peripheral Controllers are connected to the processors as slaves via the AMBA interconnect. These controllers has R/W control register that may be addressed in the memory space of the processor. The programmable logic part is also connected to interconnect as a slave and designs can implement multiple cores in the FPGA fabric.

Below tables depict the external components connected to the MIO pins of the Zybo Z7.

Table 1.3 External components connection of Zybo to MIO pins

MIO 500 3.3 V	Peripherals		
Pin	Pmod	SPI Flash	GPIO
0	JF7		
1		CS	
2		DQ0	
3		DQ1	
4		DQ2	
5		DQ3	
6		SCLK	
7			LED4
8		SCLK FB	
9	JF8		
10	JF2		
11	JF3		
12	JF4		
13	JF1		
14	JF9		
15	JF10		

Table 1.3 External components connection of Zybo to MIO pins (continued)

MIO 501 1.8V	Peripherals		
Pin	ENET 0	USB 0	SDIO 0
16	TXCK		
17	TXD0		
18	TXD1		
19	TXD2		
20	TXD3		
21	TXCTL		
22	RXCK		
23	RXD0		
24	RXD1		
25	RXD2		
26	RXD3		
27	RXCTL		
28		DATA4	
29		DIR	
30		STP	
31		NXT	
32		DATA0	
33		DATA1	
34		DATA2	
35		DATA3	
36		CLK	
37		DATA5	
38		DATA6	
39		DATA7	
40			CCLK
41			CMD

Table 1.3 External components connection of Zybo to MIO pins (continued)

MIO 501 1.8V	Peripherals		
Pin	ENET 0	USB 0	SDIO 0
42			D0
43			D1
44			D2
45			D3
46		RESETN	
47			CD
48			
49			
50			
51			
52	MDC		
53	MDIO		

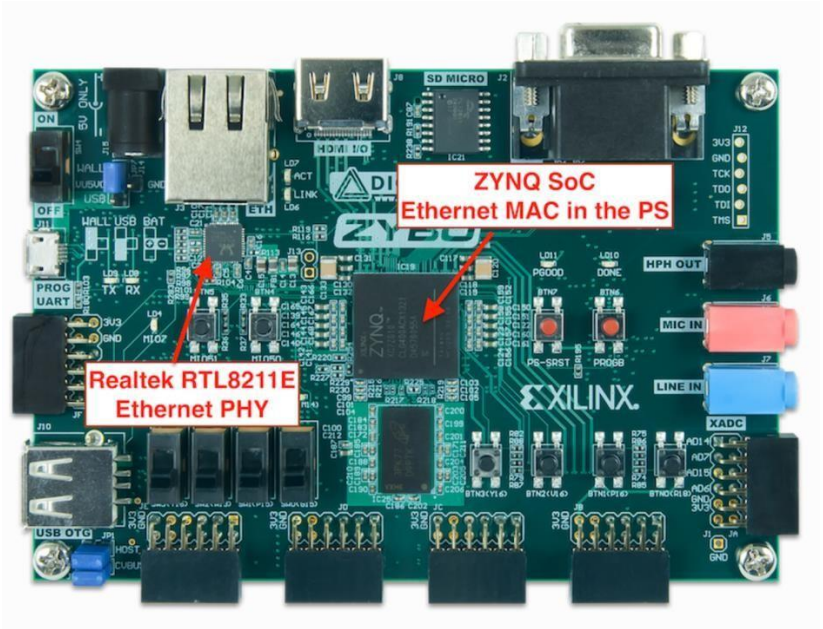
CHAPTER 2 ETHERNET

Ethernet can be explained as a technology used to connect devices in WAN or LAN. The Zybo Z7 uses Realtek RTL8211E-VL PHY to implement a 10/100/1000 Ethernet port for network connection.

2.1 Ethernet Layers

The MAC and the PHY layers comprise of an Ethernet interface. These two ethernet interface layers are implemented in separate IC's during FPGA design. In this case, MAC is implemented in PS (ZYNQ Processing System) and PHY is provided externally to PL (Programmable Logic). For ZYBO board, PHY is implemented in Realtek RTL8211E-VL IC.

Figure 2.1 Ethernet MAC and PHY in the Zybo board [03]



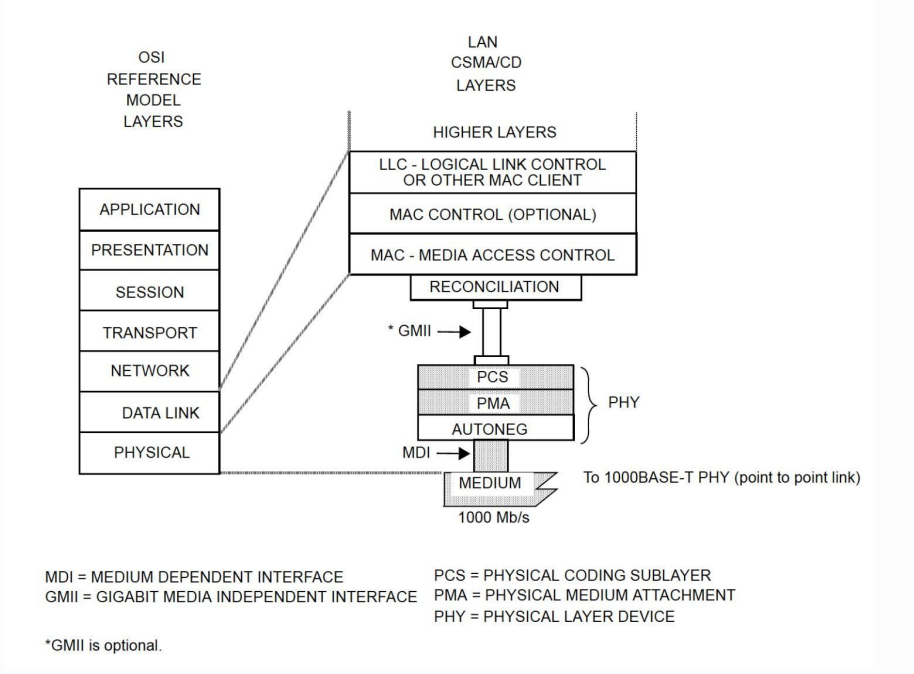
2.2 Interface between the MAC and the PHY

Once we get to know the location of MAC and PHY layers, next step is to know about their communication. Media Independent Interface commonly called as Giga Media Independent Interface is mainly responsible for communication between the MAC and PHY.

MAC implemented in PS part of the ZYNQ connects through RGMII interface.

GMII interface is provided to PL because few applications need to instantiate some PHY sublayers within PL.

Figure 2.2 Interface between MAC and the PHY



The Realtek RTL8211E-VL PHY accepts both GMII and RGMII interfaces for 1000BASE-T operation. 1000BASE-T is an Ethernet interface capable of 1Gbps data rate, it communicates through twisted-pair copper cables

The Management Data Input/Output (MDIO) interface helps The MAC is able to send and read data to and from the PHY with the help of Management Data Input/Output interface. This MDIO has a clock named Management Data Clock (MDC). PS provides outputs from the MIO

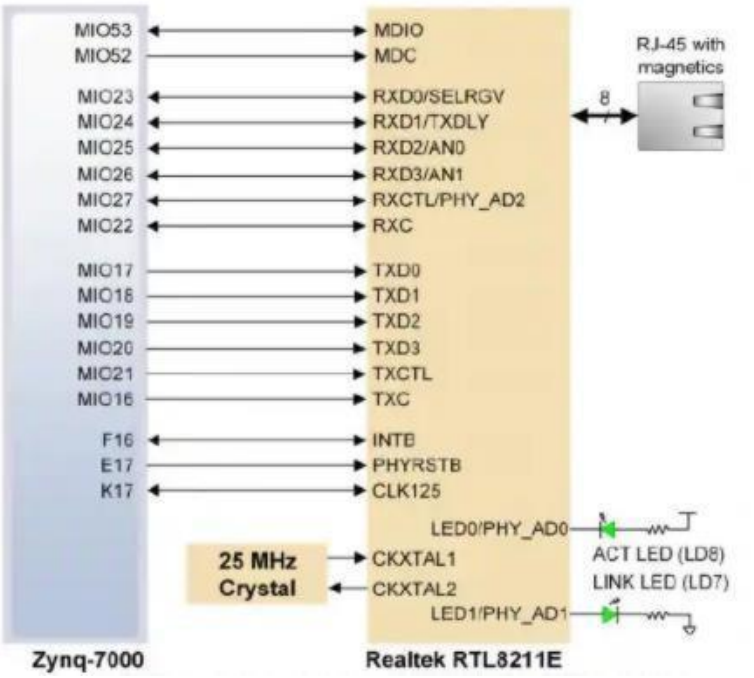
and are routed to appropriate pins (connected to the PHY) in the ZYBO board.

2.3 Ethernet PHY signals

The connection diagram is shown in Figure 3.1

After the power-up, PSY starts with Auto Negotiation enabled, advertising 10/100/1000 link speeds and full duplex. Even though Zynq is not configured, sometimes PHY is able to establish link with if an ethernet-capable partner is connected.

Figure 2.3 Ethernet PHY signals [01]



The default behavior of the two-status indicator LED's, on-board near the RJ-45 connector indicating traffic (LD8) and valid link state (LD7) are shown in Table 3.1

Table 2.1 Ethernet Status LED's

Function	Designator	State	Description
LINK	LD7	Steady on	Link 10/100/1000
		Blinking 0.4s ON, 2s OFF	Link, Energy Efficient Ethernet (EEE) mode
ACT	LD8	Blinking	Transmitting or Receiving

The Zynq incorporates two independent Gigabit Ethernet Controllers. These ethernet controllers has a capacity to implement 10 or 100 or 1000 Ethernet MAC. This MAC can be either Full or Half duplex. GEM 0 is mapped to the MIO pins where the PHY interfaces. Since the MIO bank is powered from 1.8V, RGMII interface utilizes 1.8V HSTL drivers. The Zybo Z7 Vivado board files handles the mapping of correct pins and configuration of interfaces.

The MDIO bus is available for management though the default power-up configuration of the PHY is sufficient in most applications. A 5-bit address, 00001 is assigned to the RTl8211E-VL on the MDIO bus. The status information can be read, and configurations can be changed by simple register read and write commands. The industry-standard register map is followed by the Realtek PHY for all the basic configurations.

The RGMII specification would need the receive (RXC) and transmit (TXC) clock to be delayed comparative to the data signals (RXD[0:3], RXCTL and TXD[0:3], TXCTL). These delays are also required by the Xilinx PCB guidelines. The RTL8211E-VL is capable of inserting a 2ns delay on both TXC and RXC so that the board traces need not to be made longer.

Each node requires a unique MAC address on an Ethernet network. The One Time Programmable (OTP) region of the Quad-SPI flash has been programmed with 48-bit globally unique EUI-48/64 compatible identifier. The OTP address range[0x20;0x25] contains the identifier with first byte in transmission byte order being the lowest address.

The identifier is automatically handled in the U-boot bootloader while using Peta Linux and the Linux

system is also automatically configured to use this unique MAC address. The identifier can be found on top of Zybo Z7 and also above headphone output jack.

Xilinx provides a lwip TCP/IP stack which can be automatically generated in Xilinx SDK to get started with using ethernet port on a bare-metal application. When the Zybo Z7 is used with Peta Linux generated embedded Linux system, the ethernet port will automatically appear as a network device typically named as eth0.

2.4 AXI Ethernet Subsystem

The ethernet subsystem provides an AXI4-Lite bus interface to make a successful connection to processor core. This allows access to the registers. Slave interface allows single R/O data transfers. AXI4-Stream buses provide support to TCP/UDP checksum.

A 1000BASE-X PHY is implemented in FPGA using 1G/2.5G Ethernet PCS/PMA or SGMII module is configured in 1000BASE-X mode.

AXI 1G/2.5G Ethernet subsystem may be configured in non-processor mode where buffer module is absent.

The supported physical interface types are as follows:

1. GMII: Gigabit Media Independent Interface will be defined by IEEE 802.3 specification; it provides support for Ethernet operation at 10 Mb/s, 100 Mb/s and 1Gb/s speeds.
2. MII: The Media Independent Interface will be defined by IEEE 802.3 specification; it provides support for Ethernet operation at 10Mb/s and 100 Mb/s speeds.
3. RGMII: Reduced Gigabit Media Independent Interface is a double data rate version of GMII; it supports ethernet operation at 10, 100, 1Gb/s speeds.

4. 1000BASE-X: Physical Coding Sublayer and Physical Medium Attachment operation is defined in IEEE 802.3-2008 standard.
5. GMII to Serial-GMII Bridge
6. BOTH: This feature supports switching between 1000BASE-X and SGMII standard.

The infrastructure cores used by AXI Ethernet Subsystem are:

1. Tri-Mode Ethernet MAC core
2. AXI Ethernet Buffer
3. 1G/2.5 Ethernet PCS/PMA or SGMII core
4. Timer Synchronization core

Data Integrity while using TCP or UDP Ethernet protocols, is controlled by calculating and verifying checksum values over TCP and UDP frame data. In this case, checksum information is passed between software and the subsystem with the help of AXI-4 stream control and AXI-4 stream status interface. [06]

2.5 Sending and Receiving Data through ZYNQ Ethernet

We use the hard MAC of the ZYNQ SoC to connect it to the external PHY through the RGMII interface of the MIO. This manages the configuration of this PHY chip through the MDIO.

The MAC Transmitter and receiver modules (in the PS) are connected to a FIFO, later exchanges data with memory through a DMA controller.

During Transmitting, data is fetched from memory and is written into FIFO. Then FIFO passes data on to the MAC Tx chain. Whereas MAC Rx writes data which is received into FIFO. The DMA sends this data to the memory.

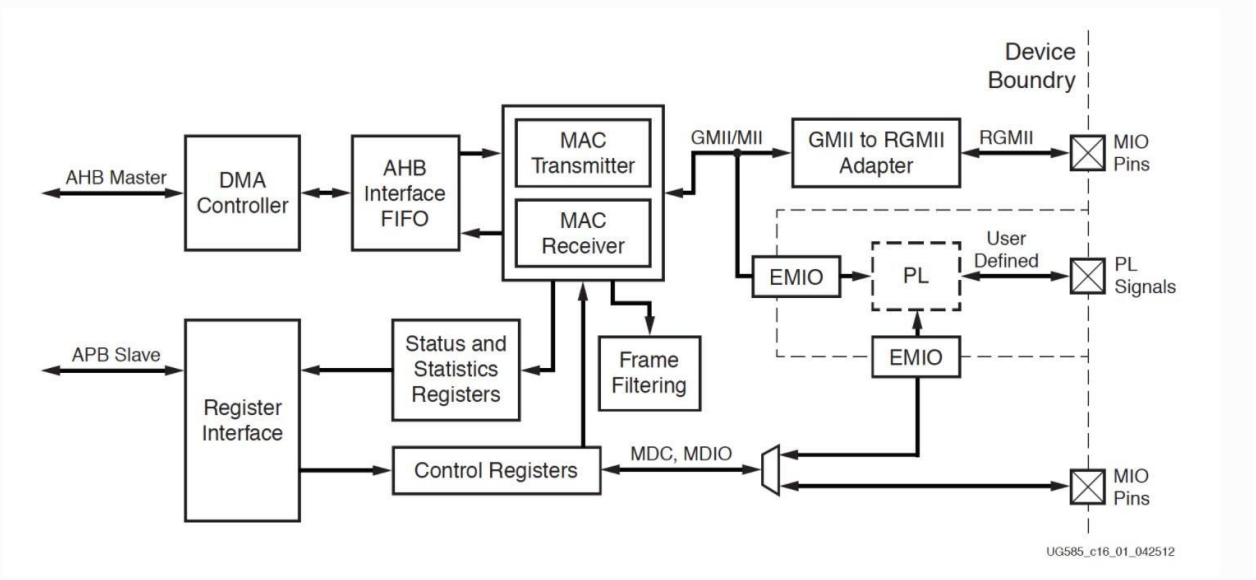


Figure 2.4 ZYNQ Ethernet Controller

CHAPTER 3 WIRESHARK

3.1 Introduction

Wireshark is an open-source software used to analyze Network Packets. Network packet analyzer represents the captured packet data, and it acts as a measuring device to examine inside functioning of a network cable. Wireshark captures the live packet data from network interface and displays it with detailed information. Wireshark can capture the data from various network types like ethernet, USB, Wireless LAN, Bluetooth, and more

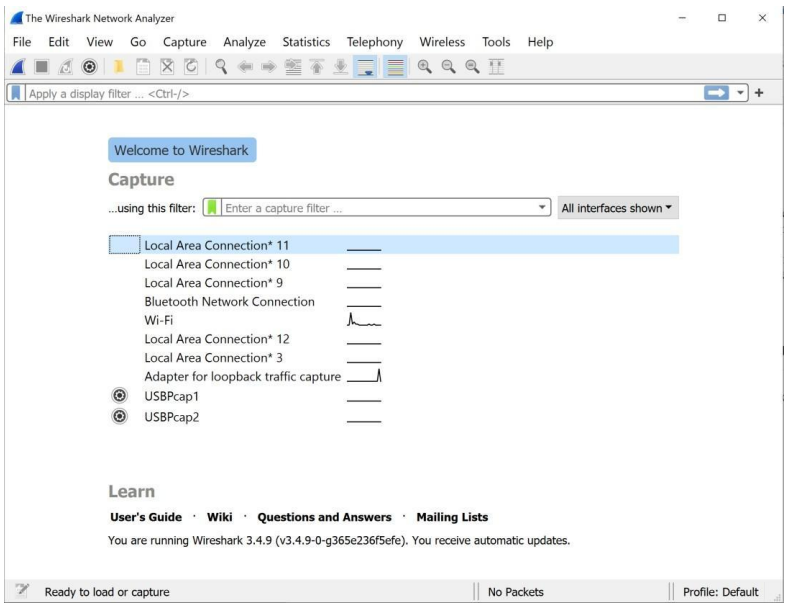
Benefits of wireshark

- Network administrators use it to troubleshoot network problems
- Network security engineers use it to examine security problems
- QA engineers use it to verify network applications
- Developers use it to debug protocol implementations
- People use it to learn network protocol internals

The following are some of the many features Wireshark provides:

- Available for UNIX and Windows.
- Capture live packet data from a network interface.
- Open files containing packet data captured with tcpdump/WinDump, Wireshark, and many other packet capture programs.
- Import packets from text files containing hex dumps of packet data.
- Display packets with very detailed protocol information.
- Save packet data captured.

Figure 3.1 Wireshark Network Analyzer



3.2 System Requirements

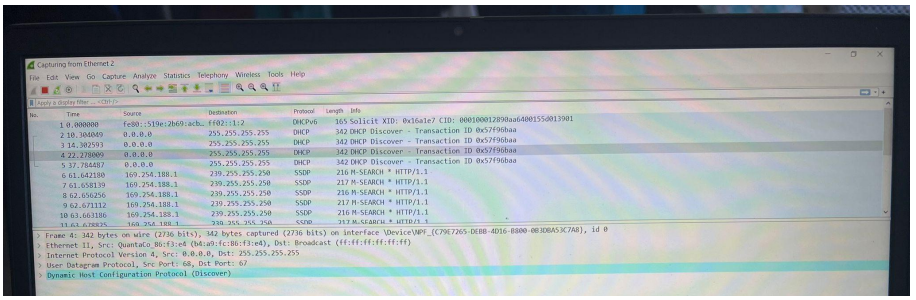
Wireshark supports Microsoft Windows, macOS, INIX, Linux and BSD. This project is done on Microsoft windows 10. It supports windows 10, 8.1, Server 2019, Server 2016, Server 2012 R2, and Server 2012.

Wireshark software requires the Universal C Runtime which is included with windows 10 and 64-bit AMD 64

3.3 The “Packet List” Pane

The packet list plane shows the packets in the current file that is being captured.

Figure 3.2 The “Packet List” Pane [04]

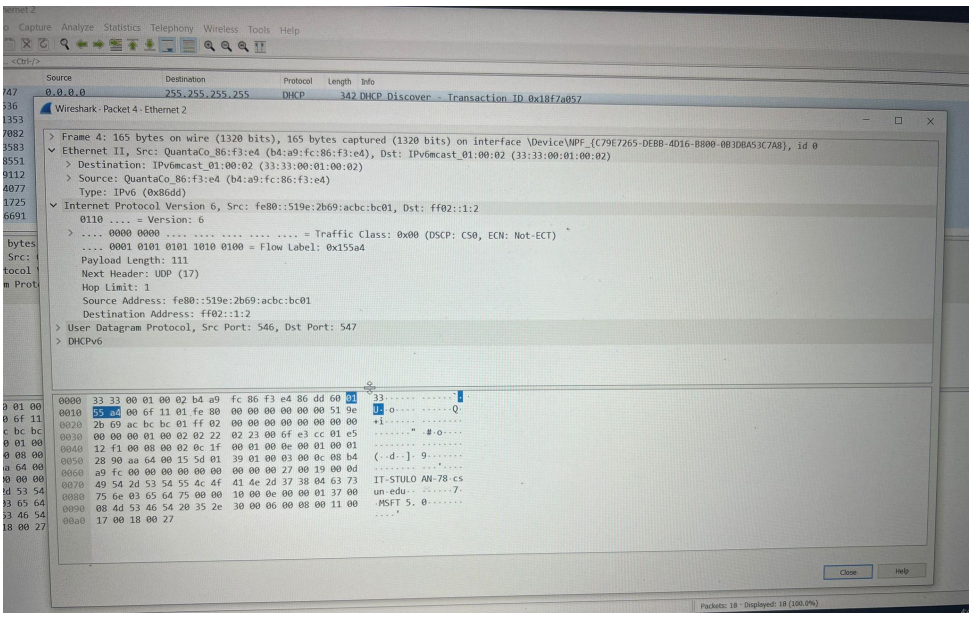


Each line represents one packet in the captured file. If you want more details to be displayed, select a line in the pane. “Packet Details” and “Packet Bytes” panes displays more details in the pane.

3.4 The “Packet Details” Pane

The packet details pane represents more details about current packet. This pane shows protocols and protocol fields.

Figure 3.3. The “Packet Details” pane


Wireshark · Packet 4 · Ethernet 2
Frame 4: 165 bytes on wire (1320 bits), 165 bytes captured (1320 bits) on interface \Device\NPF_{C79E7265-DEBB-4D16-B800-0B3DBA53C7A8}, id 0
Ethernet II, Src: QuantaCo_86:f3:e4 (b4:a9:fc:86:f3:e4), Dst: IPv6mcast_01:00:02 (33:33:00:01:00:02)
Destination: IPv6mcast_01:00:02 (33:33:00:01:00:02)
Source: QuantaCo_86:f3:e4 (b4:a9:fc:86:f3:e4)
Type: IPv6 (0x86dd)
Internet Protocol Version 6, Src: fe80::519e:2b69:acbc:bc01, Dst: ff02::1:2
0110 .... = Version: 6
.... 0000 0000 .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
.... 0001 0101 0101 1010 0100 = Flow Label: 0x155a4
Payload Length: 111
Next Header: UDP (17)
Hop Limit: 1
Source Address: fe80::519e:2b69:acbc:bc01
Destination Address: ff02::1:2
User Datagram Protocol, Src Port: 546, Dst Port: 547
DHCPv6
Close
Help

CHAPTER 4

HARDWARE DESIGN

Xilinx Vivado is a software produced by Xilinx for synthesis and analysis of HDL designs and to program FPGA. Vivado includes an built-in logic simulator. Vivado converts any language code like C , Verilog , VDHL into programmable logic.

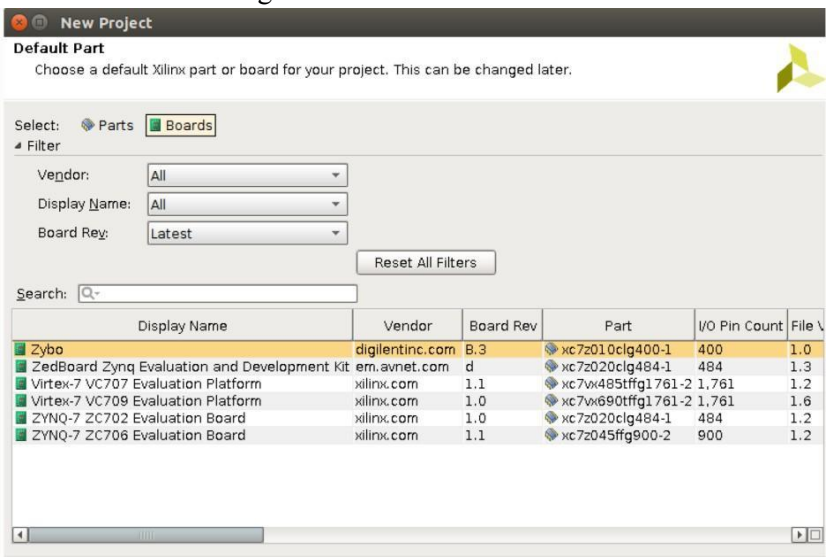
The following steps can be followed to create hardware design that implements ethernet interface on the ZYBO board.

1. Create a new project

Figure 4.1 Xilinx Vivado 2018.3 [05]

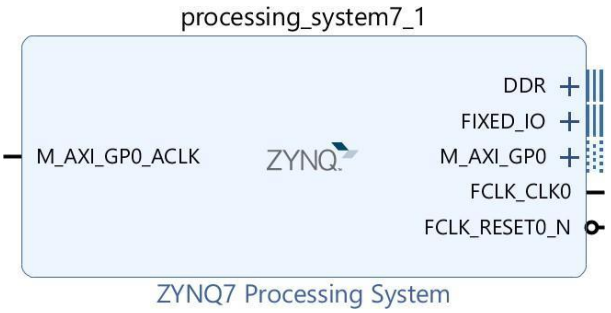
2. In the “Default Part” section, set ZYBO board as the project board

Figure 4.2 Selection of ZYBO board



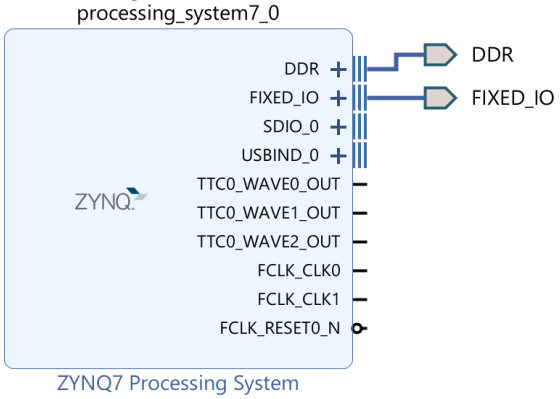
3. Create a block design

Figure 4.3 Block diagram of ZYNQ7 Processing System used in project



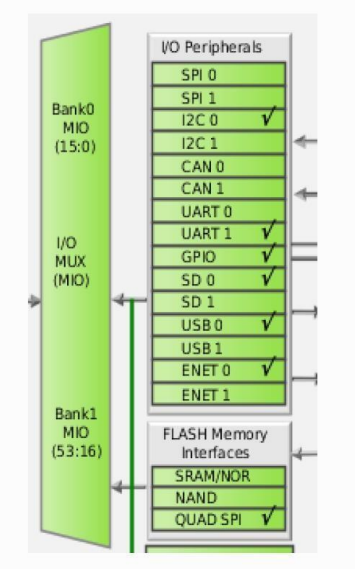
4. Run block automation

Figure 4.4 Block diagram of automated ZYNQ7 Processing System



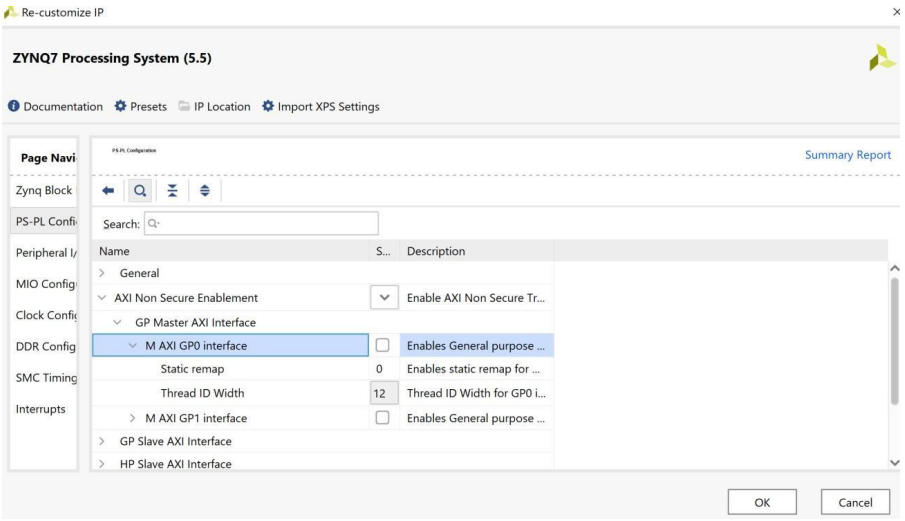
5 Note that any of the two Gigabit Ethernet controller is enabled in ENET 0 configuration:

Figure 4.5 ZYNQ Block Design with Ethernet enabled



5. Go to PS-PL configuration pane and disable the AXI master interface 0.

Figure 4.6 Disabling of AXI master interface 0



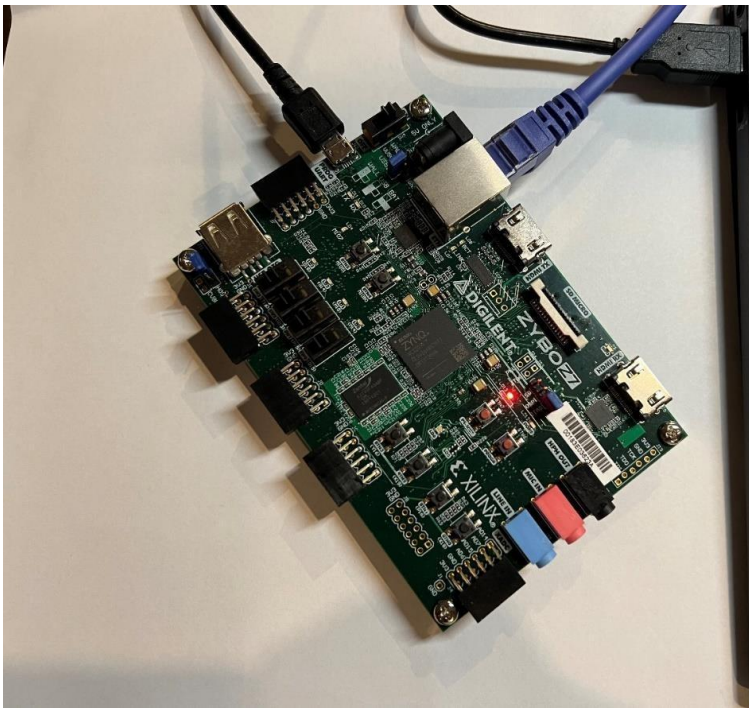
6. Save the design.
7. Validate the design.
8. Create an HDL wrapper
9. Generate bitstream.
10. File-> Export->Export hardware. Make sure to include bitstream.
11. Launch SDK.
12. Create Board Support Package.
13. Create “intra_dma” code
14. Testing

CHAPTER 5 TESTING

5.1 Connection between ZYBO and computer

To test the project, plug an ethernet cable between computer and the ZYBO board.

Figure 5.1 Testing



5.2 Capturing of packets using Wireshark software

Wireshark is a open software to display the ethernet protocol packets information in readable format.

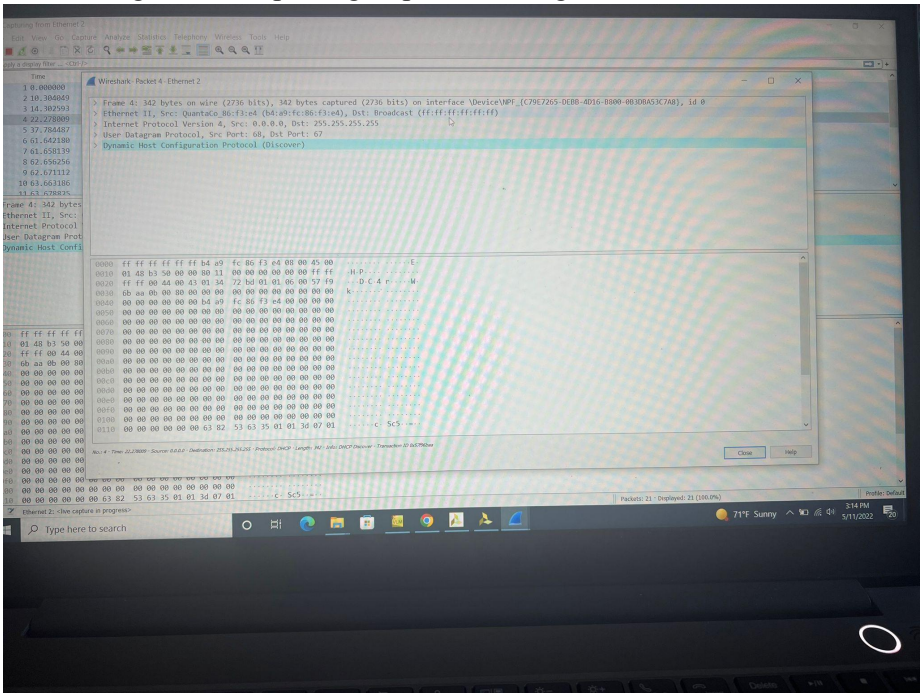
Step 1: Once the host computer is installed with Wireshark, launch the software.

Step 2: Select interface that is likely to be inspected.

Step 3: Select the start button. Once the proper connection is established, start the capture. Select the ethernet from the lists. Packets captured from Wireshark can be seen as below. Protocol UDP can be

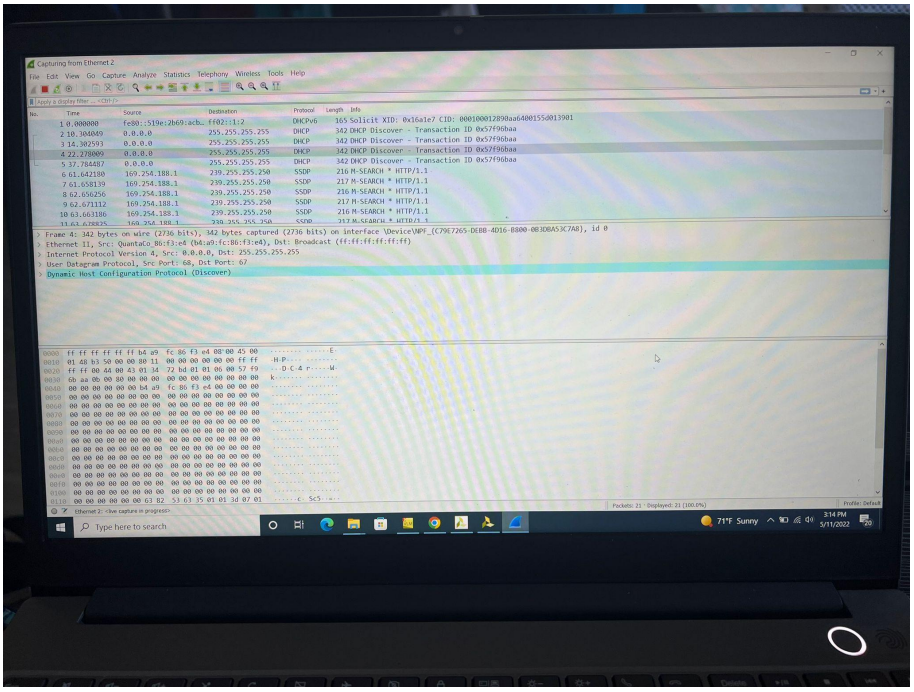
seen capturing on Wireshark from the ZYBO board.

Figure 5.2 Capturing of packets using Wireshark software



If you click on the packet received, you get complete details about the packet received.

Figure 5.3 Detailed view of captured packets using Wireshark software



As you can see in the figure above, packet received is LLC protocol-based packet. This is the transmission of data from zybo board to computer.

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

Capturing packets is completed using the Wireshark software from the scratch. Understanding ethernet and the connections is very important aspect of this project. Xilinx Vivado 2018.3 software provides all the functions to design, connect and to verify the output by connecting ZYBO board externally to the computer. Learnt about ZYBO board and variants of ZYBO Z7 boards. Capturing of the packets generated by the ZYBO board was possible with the help of Xilinx Vivado and Wireshark software.

This project can be used to communicate with Zybo board over ethernet and to capture packets sent by the ZYBO board.

6.2 Future Work

With some more research and work on the ethernet helps to connect two boards at the same time. Two boards can be connected over ethernet to check the packets. Also communication between two ZYBO boards are possible with the help of Xilinx Vivado Software.

Some changes and addition in the main code can be done that connects two boards. Specifying the port number for both the board in code makes it possible for two boards to communicate between each other.

APPENDIX A

design_1_wrapper.v code

```
//Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
//_____________________________________________________________
//Tool Version: Vivado v.2018.3 (win64) Build 2405991 Thu Dec 6 23:38:27 MST 2018
//Date        : Wed Oct 27 14:25:45 2021
//Host        : LAPTOP-UTIDMIGG running 64-bit major release (build 9200)
//Command     : generate_target design_1_wrapper.bd
//Design      : design_1_wrapper
//Purpose     : IP block netlist
//_____________________________________________________________
`timescale 1 ps / 1 ps

module design_1_wrapper
   (DDR_addr,
    DDR_ba,
    DDR_cas_n,
    DDR_ck_n,
    DDR_ck_p,
    DDR_cke,
    DDR_cs_n,
    DDR_dm,
    DDR_dq,
    DDR_dqs_n,
    DDR_dqs_p,
    DDR_odt,
    DDR_ras_n,
    DDR_reset_n,
    DDR_we_n,
    FIXED_IO_ddr_vrn,
    FIXED_IO_ddr_vrp,
    FIXED_IO_mio,
    FIXED_IO_ps_clk,
    FIXED_IO_ps_porb,
    FIXED_IO_ps_srstb);
  inout [14:0]DDR_addr;
  inout [2:0]DDR_ba;
  inout DDR_cas_n;
  inout DDR_ck_n;
  inout DDR_ck_p;
  inout DDR_cke;
  inout DDR_cs_n;
  inout [3:0]DDR_dm;
  inout [31:0]DDR_dq;
  inout [3:0]DDR_dqs_n;
  inout [3:0]DDR_dqs_p;
  inout DDR_odt;
  inout DDR_ras_n;
  inout DDR_reset_n;


  inout DDR_we_n;
```

```
inout
FIXED_IO_ddr_vrn;
inout
FIXED_IO_ddr_vrp;
inout
[53:0]FIXED_IO_mio;
inout
FIXED_IO_ps_clk;
inout
FIXED_IO_ps_porb;
inout
FIXED_IO_ps_srstb;

wire
[14:0]DDR_addr;
wire
[2:0]DDR_ba;
wire DDR_cas_n;
wire DDR_ck_n;
wire
DDR_ck_p;
wire DDR_cke;
wire
DDR_cs_n;
wire
[3:0]DDR_dm;
wire
[31:0]DDR_dq;
wire
[3:0]DDR_dqs_n;
wire
[3:0]DDR_dqs_p;
wire DDR_odt;
wire
DDR_ras_n;
wire
DDR_reset_n;
wire
DDR_we_n;
wire
FIXED_IO_ddr_vrn;
wire
FIXED_IO_ddr_vrp;
wire
[53:0]FIXED_IO_mio;
wire
FIXED_IO_ps_clk;
wire
FIXED_IO_ps_porb;
wire
FIXED_IO_ps_srstb;

design_1 design_1_i
    (.DDR_addr(DDR_addr),
     .DDR_ba(DDR_ba),
```

```
.DDR_cas_n(DDR_cas_n),
.DDR_ck_n(DDR_ck_n),
.DDR_ck_p(DDR_ck_p),
.DDR_cke(DDR_cke),
.DDR_cs_n(DDR_cs_n),
.DDR_dm(DDR_dm),
.DDR_dq(DDR_dq),
.DDR_dqs_n(DDR_dqs_n),
.DDR_dqs_p(DDR_dqs_p),
.DDR_odt(DDR_odt),
.DDR_ras_n(DDR_ras_n),
.DDR_reset_n(DDR_reset_n),
.DDR_we_n(DDR_we_n),
.FIXED_IO_ddr_vrn(FIXED_IO_ddr_vrn),
.FIXED_IO_ddr_vrp(FIXED_IO_ddr_vrp),
.FIXED_IO_mio(FIXED_IO_mio),
.FIXED_IO_ps_clk(FIXED_IO_ps_clk),
.FIXED_IO_ps_porb(FIXED_IO_ps_porb),
.FIXED_IO_ps_srstb(FIXED_IO_ps_srstb));
```

endmodule

APPENDIX B

```
//Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
//_____________________________________________________
//Tool Version: Vivado v.2018.3 (win64) Build 2405991 Thu Dec 6 23:38:27 MST 2018
//Date        : Wed Oct 27 14:25:45 2021
//Host        : LAPTOP-UTIDMIGG running 64-bit major release (build 9200)
//Command     : generate_target design_1.bd
//Design      : design_1
//Purpose     : IP block netlist
//_____________________________________________________
`timescale 1 ps / 1 ps

(* CORE_GENERATION_INFO
=
"design_1,IP_Integrator,{x_ipVendor=xilinx.com,x_ipLibrary=BlockDiagram,x_ipName=design_1,x
_ip
Version=1.00.a,x_ipLanguage=VERILOG,numBlks=1,numReposBlks=1,numNonXlnxBlks=0,numH
ier
Blks=0,maxHierDepth=0,numSysgenBlks=0,numHlsBlks=0,numHdlrefBlks=0,numPkgbdBlks=0,bds
our ce=USER,da_ps7_cnt=1,synth_mode=OOC_per_IP}" *) (* HW_HANDOFF = "design_1.hwdef"
*) module design_1
  (DDR_add
   r,
   DDR_ba,
   DDR_cas
   _n,
   DDR_ck_
   n,
   DDR_ck_
   p,
   DDR_cke,
   DDR_cs_
   n,
   DDR_dm,
   DDR_dq,
   DDR_dqs
   _n,
   DDR_dqs
   _p,
   DDR_odt,
   DDR_ras_
   n,
   DDR_rese
   t_n,
   DDR_we_
   n,
   FIXED_IO_ddr_v
   rn,
   FIXED_IO_ddr_v
   rp,
```

```
    FIXED_IO_mio,
    FIXED_IO_ps_cl
    k,
    FIXED_IO_ps_po
    rb,
    FIXED_IO_ps_sr
    stb);
  (* X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR ADDR" *) (*
X_INTERFACE_PARAMETER = "XIL_INTERFACENAME DDR,
AXI_ARBITRATION_SCHEME TDM, BURST_LENGTH 8, CAN_DEBUG false,
CAS_LATENCY 11, CAS_WRITE_LATENCY 11, CS_ENABLED true, DATA_MASK_ENABLED
true, DATA_WIDTH 8, MEMORY_TYPE COMPONENTS, MEM_ADDR_MAP
ROW_COLUMN_BANK, SLOT Single, TIMEPERIOD_PS
1250" *) inout [14:0]DDR_addr;
  (* X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR BA" *) inout [2:0]DDR_ba;
  (* X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR CAS_N" *) inout DDR_cas_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR CK_N" *) inout DDR_ck_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR CK_P" *) inout DDR_ck_p; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR CKE" *) inout DDR_cke; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR CS_N" *) inout DDR_cs_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR DM" *) inout [3:0]DDR_dm; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR DQ" *) inout [31:0]DDR_dq; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR DQS_N" *) inout [3:0]DDR_dqs_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR DQS_P" *) inout [3:0]DDR_dqs_p; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR ODT" *) inout DDR_odt; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR RAS_N" *) inout DDR_ras_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR RESET_N" *) inout DDR_reset_n; (*
  X_INTERFACE_INFO = "xilinx.com:interface:ddrx:1.0 DDR WE_N" *) inout DDR_we_n; (*
  X_INTERFACE_INFO = "xilinx.com:display_processing_system7:fixedio:1.0 FIXED_IO DDR_VRN" *) (*
  X_INTERFACE_PARAMETER = "XIL_INTERFACENAME FIXED_IO, CAN_DEBUG false" *) inout
  FIXED_IO_ddr_vrn; (* X_INTERFACE_INFO = "xilinx.com:display_processing_system7:fixedio:1.0
  FIXED_IO DDR_VRP" *) inout FIXED_IO_ddr_vrp; (* X_INTERFACE_INFO =
  "xilinx.com:display_processing_system7:fixedio:1.0 FIXED_IO MIO" *) inout [53:0]FIXED_IO_mio; (*
  X_INTERFACE_INFO = "xilinx.com:display_processing_system7:fixedio:1.0 FIXED_IO PS_CLK" *) inout
  FIXED_IO_ps_clk; (* X_INTERFACE_INFO = "xilinx.com:display_processing_system7:fixedio:1.0
  FIXED_IO PS_PORB" *) inout FIXED_IO_ps_porb; (* X_INTERFACE_INFO =
  "xilinx.com:display_processing_system7:fixedio:1.0 FIXED_IO PS_SRSTB" *) inout FIXED_IO_ps_srstb;
```

```
 wire [14:0] processing_system7_0_DDR_ADDR;
 wire [2:0] processing_system7_0_DDR_BA;
 wire processing_system7_0_DDR_CAS_N.
 wire processing_system7_0_DDR_CKE;
 wire processing_system7_0_DDR_CK_N;
 wire processing_system7_0_DDR_CK_P;
 wire processing_system7_0_DDR_CS_N;
 wire [3:0]processing_system7_0_DDR_DM;
 wire [31:0]processing_system7_0_DDR_DQ;
 wire [3:0]processing_system7_0_DDR_DQS_N;
 wire [3:0]processing_system7_0_DDR_DQS_P;
 wire processing_system7_0_DDR_ODT;
 wire processing_system7_0_DDR_RAS_N;
 wire processing_system7_0_DDR_RESET_N;
 wire processing_system7_0_DDR_WE_N;
 wire processing_system7_0_FIXED_IO_DDR_VRN;
 wire processing_system7_0_FIXED_IO_DDR_VRP;
 wire [53:0]processing_system7_0_FIXED_IO_MIO;
 wire processing_system7_0_FIXED_IO_PS_CLK;
 wire processing_system7_0_FIXED_IO_PS_PORB;
 wire processing_system7_0_FIXED_IO_PS_SRSTB;
  design_1_processing_system7_0_0
      processing_system7_0
      (.DDR_Addr(DDR_addr[14:0]),
       .DDR_BankAddr(DDR_ba[2:0]),
       .DDR_CAS_n(DDR_cas_n),
       .DDR_CKE(DDR_cke),
       .DDR_CS_n(DDR_cs_n),
       .DDR_Clk(DDR_ck_p),
       .DDR_Clk_n(DDR_ck_n),
       .DDR_DM(DDR_dm[3:0]),
       .DDR_DQ(DDR_dq[31:0]),
       .DDR_DQS(DDR_dqs_p[3:0]),
       .DDR_DQS_n(DDR_dqs_n[3:0]),
       .DDR_DRSTB(DDR_reset_n),
       .DDR_ODT(DDR_odt),
       .DDR_RAS_n(DDR_ras_n),
       .DDR_VRN(FIXED_IO_ddr_vrn),
       .DDR_VRP(FIXED_IO_ddr_vrp),
       .DDR_WEB(DDR_we_n),
       .MIO(FIXED_IO_mio[53:0]),
       .PS_CLK(FIXED_IO_ps_clk),
       .PS_PORB(FIXED_IO_ps_porb),
       .PS_SRSTB(FIXED_IO_ps_srstb),
       .SDIO0_WP(1'b0),
       .USB0_VBUS_PWRFAULT(1'b0));
endmodule
```

APPENDIX C

Intra_dma ethernet code

```
#include
"xemacps_example.h"
#include "xil_exception.h"
#ifndef_MICROBLAZE
_____
#include "xil_mmu.h"
#endif
#ifdef_MICROBLAZE____
#define XPS_SYS_CTRL_BASEADDRXPAR_PS7_SLCR_0_S_AXI_BASEADDR
#endif
#ifdef
XPAR_INTC_0_DEVICE_ID
#define INTC              XIntc
#define
EMACPS_DEVICE_IDXPAR_XEMACPS_0_DEVICE_ID
#define INTC_DEVICE_ID
XPAR_INTC_0_DEVICE_ID
#else
#define INTC              XScuGic
#define EMACPS_DEVICE_IDXPAR_XEMACPS_0_DEVICE_ID
#define INTC_DEVICE_ID                XPAR_SCUGIC_SINGLE_DEVICE_ID
#endif

#if
defined(XPAR_INTC_0_DEVICE_ID
) #define EMACPS_IRPT_INTR
        XPAR_AXI_INTC_0_PROCESSING_SYSTEM7_0_IRQ_P2F_ENET0_INTR
#elif defined(XPAR_PSU_ETHERNET_0_DEVICE_ID) ||

#define RXBD_CNT      32      /* Number of RxBDs to use */
#define TXBD_CNT      32      /* Number of TxBDs to use */
/************************** Variable Definitions
**************************/ EthernetFrame TxFrame;     /* Transmit buffer */
EthernetFrame RxFrame;                /* Receive buffer */

/*
 * Buffer descriptors are allocated in uncached memory. The memory is made
 * uncached by setting the attributes appropriately in the MMU table.
```

```
 */
#define RXBD_SPACE_BYTES
XEmacPs_BdRingMemCalc(XEMACPS_BD_ALIGNMENT, RXBD_CNT)
#define TXBD_SPACE_BYTES XEmacPs_BdRingMemCalc(XEMACPS_BD_ALIGNMENT,
TXBD_CNT)


/*
 * Buffer descriptors are allocated in uncached memory. The memory is made
 * uncached by setting the attributes appropriately in the MMU table.
 */
#define RX_BD_LIST_START_ADDRESS
0x0FF00000 #define TX_BD_LIST_START_ADDRESS

0x0FF70000

#define FIRST_FRAGMENT_SIZE 64

/*
 * Counters to be incremented by callbacks
 */
volatile s32 FramesRx;              /* Frames have been
received */ volatile s32 FramesTx;      /* Frames have been
sent */
volatile s32 DeviceErrors;          /* Number of errors detected in the

device */ u32 TxFrameLength;

#ifndef
TESTAPP_GEN
static              INTC
IntcInstance; #endif

#ifdef__ICCARM____
#pragma data_alignment =
64 XEmacPs_Bd
BdTxTerminate;
XEmacPs_Bd BdRxTerminate;
```

```
#pragma data_alignment
= 4 #else
XEmacPs_Bd BdTxTerminate__attribute___((aligned(64)));

XEmacPs_Bd BdRxTerminate__attribute___
((aligned(64))); #endif

u32
GemVersion;
u32 Platform;

/************************** Function Prototypes ****************************/

/*
 * Example
 */
LONG EmacPsDmaIntrExample(INTC *IntcInstancePtr,
                          XEmacPs *EmacPsInstancePtr,
                          u16 EmacPsDeviceId, u16 EmacPsIntrId);

LONG EmacPsDmaSingleFrameIntrExample(XEmacPs * EmacPsInstancePtr);

/*
 * Interrupt setup and Callbacks for examples
 */

static LONG EmacPsSetupIntrSystem(INTC * IntcInstancePtr,
                                  XEmacPs *
                                  EmacPsInstancePtr, u16
                                  EmacPsIntrId);

static void EmacPsDisableIntrSystem(INTC * IntcInstancePtr,
                                    u16 EmacPsIntrId);

static void XEmacPsSendHandler(void
*Callback); static void
XEmacPsRecvHandler(void *Callback);
static void XEmacPsErrorHandler(void *Callback, u8 direction, u32 word);

/*
 * Utility routines
 */
static LONG EmacPsResetDevice(XEmacPs * EmacPsInstancePtr);
void XEmacPsClkSetup(XEmacPs *EmacPsInstancePtr, u16 EmacPsIntrId);
void XEmacPs_SetMdioDivisor(XEmacPs *InstancePtr, XEmacPs_MdcDiv Divisor);
/*****************************************************************************/
/**
*
* This is the main function for the EmacPs example.
******************************************************************************
**/ #ifndef TESTAPP_GEN
int main(void)
{
        LONG Status;
```

```
        xil_printf("Entering into main() \r\n");

        /*
         * Call the EmacPs DMA interrupt example , specify the parameters
         * generated in xparameters.h
         */
          Status = EmacPsDmaIntrExample(&IntcInstance,
                                        &EmacPsInstance,
                                        EMACPS_DEVICE_ID,
                                        EMACPS_IRPT_INTR);

          if (Status != XST_SUCCESS) {
                  EmacPsUtilErrorTrap("Emacps intr dma Example Failed\r\n");
                  return XST_FAILURE;
          }

          xil_printf("Successfully ran Emacps intr dma Example\r\n");
          return XST_SUCCESS;
}
#endif

          XEmacPs_BdClear(&BdTemplate);
          XEmacPs_BdSetStatus(&BdTemplate, XEMACPS_TXBUF_USED_MASK);
                                                                          Status =
        /*
         * Create the TxBD ring
         */
        Status = XEmacPs_BdRingCreate(&(XEmacPs_GetTxRing
                                        (EmacPsInstancePtr)),
                                        (UINTPTR)
                                        TX_BD_LIST_START_ADDRESS,
                                        (UINTPTR)
                                        TX_BD_LIST_START_ADDRESS,
                                        XEMACPS_BD_ALIGNMENT,
                                        TXBD_CNT);
        if (Status != XST_SUCCESS) {
                EmacPsUtilErrorTrap
                        ("Error setting up TxBD space,
                BdRingCreate"); return XST_FAILURE;
        }
        Status = XEmacPs_BdRingClone(&(XEmacPs_GetTxRing(EmacPsInstancePtr)),
                                        &BdTemplate,
        XEMACPS_SEND); if (Status != XST_SUCCESS) {
                EmacPsUtilErrorTrap
                        ("Error setting up TxBD space,
                BdRingClone"); return XST_FAILURE;
        }

        if (GemVersion > 2)
        {
                XEmacPs_BdClear(&BdRxTerminate);
```

```
                XEmacPs_BdSetAddressRx(&BdRxTerminate, (XEMACPS_RXBUF_NEW_MASK |
                                                XEMACPS_RXBUF_WRAP_MASK));
                XEmacPs_Out32((Config->BaseAddress + XEMACPS_RXQ1BASE_OFFSET),
                                (UINTPTR)&BdRxTerminate);
                XEmacPs_BdClear(&BdTxTerminate);
                XEmacPs_BdSetStatus(&BdTxTerminate, (XEMACPS_TXBUF_USED_MASK |
                                                XEMACPS_TXBUF_WRAP_MASK));
                XEmacPs_Out32((Config->BaseAddress + XEMACPS_TXQBASE_OFFSET),
                                (UINTPTR)&BdTxTerminate);
                if (Config->IsCacheCoherent == 0) {
                        Xil_DCacheFlushRange((UINTPTR)(&BdTxTerminate), 64);
                }
        }

        /*
         * Set emacps to phy loopback
         */
        if (GemVersion == 2)
        {
                XEmacPs_SetMdioDivisor(EmacPsInstancePtr, MDC_DIV_224);
                EmacpsDelay(1);
                EmacPsUtilEnterLoopback(EmacPsInstancePtr, EMACPS_LOOPBACK_SPEED_1G);
                XEmacPs_SetOperatingSpeed(EmacPsInstancePtr,
EMACPS_LOOPBACK_SPEED_1G);
        }
        else
        {
                XEmacPs_SetMdioDivisor(EmacPsInstancePtr, MDC_DIV_224);
                if ((Platform & PLATFORM_MASK) == PLATFORM_SILICON) {
                        EmacPsUtilEnterLoopback(EmacPsInstancePtr,
EMACPS_LOOPBACK_SPEED_1G);

        XEmacPs_SetOperatingSpeed(EmacPsInstancePtr,EMACPS_LOOPBACK_SPEED_1G);
                } else {
                        EmacPsUtilEnterLoopback(EmacPsInstancePtr,
EMACPS_LOOPBACK_SPEED);

        XEmacPs_SetOperatingSpeed(EmacPsInstancePtr,EMACPS_LOOPBACK_SPEED);
                }
        }



    /*
     * Setup the interrupt controller and enable interrupts
     */
    Status = EmacPsSetupIntrSystem(IntcInstancePtr,
                                        EmacPsInstancePtr, EmacPsIntrId);

    /*
     * Run the EmacPs DMA Single Frame Interrupt example
     */
```

```
	Status =
	EmacPsDmaSingleFrameIntrExample(EmacPsInstancePtr); if
	(Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

	/*
	 * Disable the interrupts for the EmacPs device
	 */
	EmacPsDisableIntrSystem(IntcInstancePtr, EmacPsIntrId);

	/*
	 * Stop the device
	 */
	XEmacPs_Stop(EmacPsInstancePt
	r);

	return XST_SUCCESS;
}


/*****************************************************************************/
/**
*
* This function sets up the clock divisors for 1000Mbps.
*
*****************************************************************************
***/ void XEmacPsClkSetup(XEmacPs *EmacPsInstancePtr, u16 EmacPsIntrId)
{
	u32 ClkCntrl;
	if (GemVersion == 2)
	{
		/***********************************/
		/* Setup device for first-time usage */
		/***********************************/

	/* SLCR unlock */
	*(volatile unsigned int *)(SLCR_UNLOCK_ADDR) =
SLCR_UNLOCK_KEY_VALUE; #ifndef__MICROBLAZE___
	if (EmacPsIntrId == XPS_GEM0_INT_ID) {
#ifdef XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV0
		/* GEM0 1G clock
		configuration*/ ClkCntrl =
		*(volatile unsigned int
		*)(SLCR_GEM0_CLK_CTRL_ADDR); ClkCntrl &=
		EMACPS_SLCR_DIV_MASK;
		ClkCntrl |= (XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV1 <<
		20); ClkCntrl |= (XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV0
		<< 8);
		*(volatile unsigned int *)(SLCR_GEM0_CLK_CTRL_ADDR) =
							ClkCntrl;
#endif
	} else if (EmacPsIntrId == XPS_GEM1_INT_ID) {
#ifdef XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV1
```

```
                    /* GEM1 1G clock configuration*/
ClkCntrl = *(volatile unsigned int *)(SLCR_GEM1_CLK_CTRL_ADDR);
 ClkCntrl &= EMACPS_SLCR_DIV_MASK; ClkCntrl |=
(XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV1 << 20);
 ClkCntrl |= (XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV0 << 8);
*(volatile unsigned int *)(SLCR_GEM1_CLK_CTRL_ADDR) = ClkCntrl;
#endif
 }
 #else
  #ifdef XPAR_AXI_INTC_0_PROCESSING_SYSTEM7_0_IRQ_P2F_ENET0_INTR
          if (EmacPsIntrId ==
  XPAR_AXI_INTC_0_PROCESSING_SYSTEM7_0_IRQ_P2F_ENET0_INT
  R) { #ifdef XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV0
                    /* GEM0 1G clock
                    configuration*/ ClkCntrl =
                    *(volatile unsigned int
                    *)(SLCR_GEM0_CLK_CTRL_ADDR); ClkCntrl &=
                    EMACPS_SLCR_DIV_MASK;
                    ClkCntrl |= (XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV1 <<
                    20); ClkCntrl |= (XPAR_PS7_ETHERNET_0_ENET_SLCR_1000MBPS_DIV0
                    << 8);
                    *(volatile unsigned int *)(SLCR_GEM0_CLK_CTRL_ADDR) =
                                                            ClkCntrl;
  #endif
          }
  #endif
  #ifdef XPAR_AXI_INTC_0_PROCESSING_SYSTEM7_1_IRQ_P2F_ENET1_INTR
          if (EmacPsIntrId ==
  XPAR_AXI_INTC_0_PROCESSING_SYSTEM7_1_IRQ_P2F_ENET1_INT
  R) { #ifdef XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV1
                    /* GEM1 1G clock
                    configuration*/ ClkCntrl =
                    *(volatile unsigned int
                    *)(SLCR_GEM1_CLK_CTRL_ADDR); ClkCntrl &=
                    EMACPS_SLCR_DIV_MASK;
                    ClkCntrl |= (XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV1 <<
                    20); ClkCntrl |= (XPAR_PS7_ETHERNET_1_ENET_SLCR_1000MBPS_DIV0
                    << 8);
                    *(volatile unsigned int *)(SLCR_GEM1_CLK_CTRL_ADDR) =
                                                            ClkCntrl;
  #endif

  #endif #endif
```

```
}


 /* SLCR lock */
 *(unsigned int *)(SLCR_LOCK_ADDR) = SLCR_LOCK_KEY_VALUE;
 #ifndef_MICROBLAZE____
 sleep(1);
 #else
 unsigned long count=0;
 while(count < 0xffff)
 {
                count++;
#endif
        }
        if ((GemVersion > 2) && ((Platform & PLATFORM_MASK) == PLATFORM_SILICON)) {
  #ifdef XPAR_PSU_ETHERNET_0_DEVICE_ID
                if (EmacPsIntrId == XPS_GEM0_INT_ID) {
                        /* GEM0 1G clock
                        configuration*/ ClkCntrl =
                        *(volatile unsigned int
                        *)(CRL_GEM0_REF_CTRL); ClkCntrl &=
                        ~CRL_GEM_DIV_MASK;
                        ClkCntrl |=
                        CRL_GEM_1G_DIV1;
                        ClkCntrl |=
                        CRL_GEM_1G_DIV0;
                        *(volatile unsigned int *)(CRL_GEM0_REF_CTRL) =
                                                                ClkCntrl
}
#endif #ifdef XPAR_PSU_ETHERNET_1_DEVICE_ID
 if (EmacPsIntrId == XPS_GEM1_INT_ID)
{ /* GEM1 1G clock configuration*/
 ClkCntrl = *(volatile unsigned int *)(CRL_GEM1_REF_CTRL);
 ClkCntrl &= ~CRL_GEM_DIV_MASK;
ClkCntrl |= CRL_GEM_1G_DIV1;
ClkCntrl |= CRL_GEM_1G_DIV0;
*(volatile unsigned int *)(CRL_GEM1_REF_CTRL) = ClkCntrl;
```

```
#ifdef XPAR_PSU_ETHERNET_2_DEVICE_ID
                if (EmacPsIntrId == XPS_GEM2_INT_ID) {

                        /* GEM2 1G clock
                        configuration*/ ClkCntrl =
                        *(volatile unsigned int
                        *)(CRL_GEM2_REF_CTRL); ClkCntrl &=
                        ~CRL_GEM_DIV_MASK;
                        ClkCntrl |=
                        CRL_GEM_1G_DIV1;
                        ClkCntrl |=
                        CRL_GEM_1G_DIV0;
                        *(volatile unsigned int *)(CRL_GEM2_REF_CTRL) =
                                                                ClkCntrl;

                }
#endif
#ifdef XPAR_PSU_ETHERNET_3_DEVICE_ID
                if (EmacPsIntrId == XPS_GEM3_INT_ID) {
                        /* GEM3 1G clock
                        configuration*/ ClkCntrl =
                        *(volatile unsigned int
                        *)(CRL_GEM3_REF_CTRL); ClkCntrl &=
                        ~CRL_GEM_DIV_MASK;
#endif
        }
}
```

Project demo link

https://drive.google.com/file/d/1SUqVcYa1iKXKoQvQFOTG6kd9Kivh-zAl/view?usp=sharing

REFERENCES

[01] Zybo Z7 Board Reference Manual Revised February 21, 2018 This manual applies to the ZyboZ7 rev. B https://cdn-reichelt.de/documents/datenblatt/A300/DIGILENT_471-014.pdf

[02] Digilent Zybo Z7 Reference Manual, Vivado Design Suite PG138 (v7.2) September 20, 2021 https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual

[03] Getting started with ZYNQ Ethernet using the Zybo board, November 19, 2016, Igor Freire https://igorfreire.com.br/2016/11/19/zynq-ethernet-interface-zybo-board/#Testing

[04] Wireshark User's Guide https://www.wireshark.org/docs/wsug_html_chunked/index.html

[05] Getting Started with Vivado, Digilant website https://digilent.com/reference/vivado/getting_started/start

[06] AXI 1G/2.5G Ethernet Subsystem, Vivado Design Suite PG138 (v7.2) September 20, 2021 https://www.xilinx.com/products/intellectual-property/axi_ethernet.html#documentation