XAPP1306 (v1.0) April 3, 2017

# PS and PL-Based Ethernet Performance with LightWeight IP Stack

Authors: Bhargav Shah, Naveen Kumar Gaddipati, Akhilesh Mahajan, and Srini Gaddam

## Summary

Lightweight IP (lwIP) is an open source TCP/IP networking stack for embedded systems. The Xilinx® software development kit (SDK) provides lwIP software customized to run on the flagship ARM® Cortex®-A53 64-bit quad-core processor or Cortex-R5 32-bit dual-core processor which is a part of the Zynq® UltraScale+™ MPSoC. This document describes how to use the lwIP library to add networking capability to embedded systems based on the Zynq UltraScale+ MPSoC. The lwIP is used to develop the echo server, web server, trivial file transfer protocol (TFTP) server, and receive and transmit performance test applications. It includes throughput numbers for PS Ethernet, PL Ethernet (1G), and PS-PL Ethernet using gigabyte Ethernet controller (GEM) for lwIP.

Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

## Introduction

lwIP is an open source networking stack designed for embedded systems. It is provided under a Berkeley Software Distribution (BSD) style license. The objective of this application note is to describe how to use lwIP shipped along with the Xilinx SDK to add networking capability to an embedded system. In general, this application note describes how applications such as an echo server or a web server can be written using lwIP [Ref 1].

Zynq UltraScale+ devices integrate an ARM flagship Cortex-A53 64-bit quad-core processor, Cortex-R5 dual-core real-time processor in PS, and PL in a single device.

The PL includes the programmable logic, configuration logic, and associated embedded functions. The PS comprises the ARM Cortex-A53 MPCore CPUs unit, Cortex-R5 processors, on-chip memory, external memory interfaces, cache coherent interconnect (CCI), and peripheral connectivity interfaces. The PS is equipped with four GEMs. For external PHY communication, the RGMII interface is routed through MIO pins, and other interfaces are routed using the EMIO interface.

In the designs described in this application note, the PS-GEM3 is connected to the TI PHY through the reduced gigabit media independent interface (RGMII). This is the default setup for the ZCU102 board. This application note focuses on the design of additional Ethernet ports. The designs described in this application note are listed below.

- PS Ethernet (GEM3) connected to a 1G physical interface in the PS through an MIO interface. See Using PS GEM through MIO.
- PS Ethernet (GEM0) that is connected to a 1000BASE-X physical interface in the PL through an EMIO interface. See Using PS GEM through EMIO.
- Ethernet implemented as soft logic in the PL (MAC) and connected to the 1000BASE-X physical interface in the PL. See Using PL 1G Ethernet.

***Note:*** GEM1 or GEM2 can also be used for PS Ethernet. The hardware design varies depending on the GEM selected.

Figure 1 shows the various Ethernet implementations on the ZCU102 board.

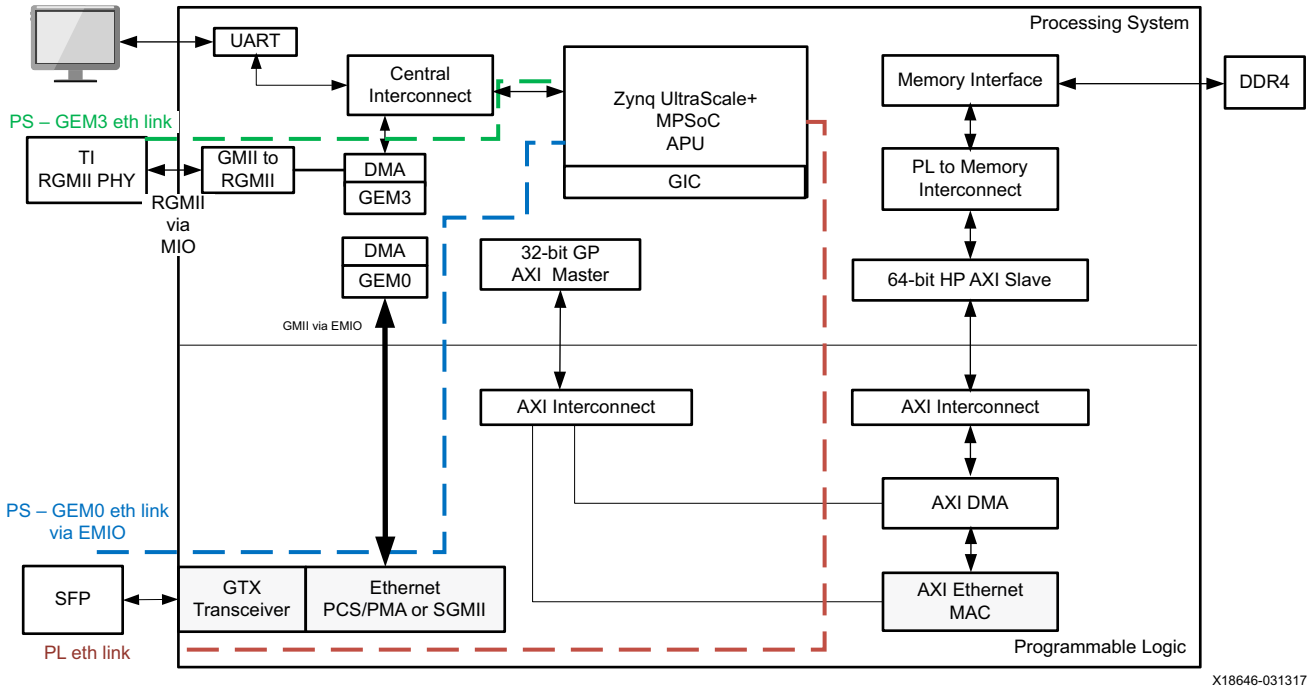


*Figure 1:* **Zynq UltraScale+ MPSoC Ethernet Interface**

***Note:*** The PS-GEM3 is always tied to the TI RGMII PHY on ZCU102 board. The 1000BASE-X PHY and the GTX transceiver are a part of the AXI Ethernet core for 1G PL Ethernet link which uses AXI 1G/2.5G Ethernet subsystem IP core [Ref 2]. The PS-PL Ethernet uses PS-GEM0 and 1G/2.5G Ethernet PCS/PMA or SGMII core [Ref 3].

# lwIP Software Applications

The reference design includes the software applications listed below. These applications are available in both the RAW and socket modes.

- Echo server
- Web server
- TFTP server

- TCP/UDP RX throughput test
- TCP/UDP TX throughput test

## Echo Server

The echo server is a simple program that echoes the input that it receives through the network. This application provides a good starting point for investigating how to write lwIP applications.

The socket mode echo server is structured as follows.

1. A main thread listens continually on a specified echo server port.
2. For each connection request, it spawns a separate echo service thread.
3. It then continues listening on the echo port.

```
while (1) {
     new_sd = lwip_accept(sock, (struct sockaddr *)&remote, &size);
     sys_thread_new(process_echo_request, (void*)new_sd, DEFAULT_THREAD_PRIO);
}
```

The echo service thread receives a new socket descriptor as its input on which it can read received data. This thread does the actual echoing of the input to the originator.

```
while (1) {
       /* read a max of RECV_BUF_SIZE bytes from socket */
       n = lwip_read(sd, recv_buf, RECV_BUF_SIZE));
       /* handle request */
       nwrote = lwip_write(sd, recv_buf, n));
}
```

***Note:*** These code snippets are not complete and are intended to show the major structure of the code only.

The socket mode provides a simple API that blocks on-socket reads and writes until they are complete. However, the socket API requires many pieces to achieve this, including a simple multi-threaded kernel. This API contains significant overhead for all operations and is therefore slow.

The RAW API provides a callback style interface to the application. For applications using the RAW API register callback, these functions are called on significant events such as accept, read, or write. A RAW API-based echo server is single-threaded and all the work is done in the callback functions. The main application loop is structured as follows.

```
while (1) {
       if (TcpFastTmrFlag) {
       tcp_fasttmr();
       TcpFastTmrFlag = 0;
       }
       if (TcpSlowTmrFlag) {
       tcp_slowtmr();
       TcpSlowTmrFlag = 0;
       }
       xemacif_input(netif);
       transfer_data();
}
```

The TcpFastTmrFlag and TcpSlowTmrFlag are required for TCP TX handling and are set in the Timer handler for every 250 ms and 500 ms, respectively.

The function of the application loop is to receive packets constantly (xemacif_input), then pass them on to lwIP. Before entering this loop, the echo server sets up certain callbacks.

```
/* create new TCP PCB structure */
pcb = tcp_new();

/* bind to specified @port */
err = tcp_bind(pcb, IP_ADDR_ANY, port);

/* we do not need any arguments to callback functions */
tcp_arg(pcb, NULL);

/* listen for connections */
pcb = tcp_listen(pcb);

/* specify callback to use for incoming connections */
tcp_accept(pcb, accept_callback);
```

This sequence of calls creates a TCP connection and sets up a callback on a connection being accepted. When a connection request is accepted, the function accept_callback is called asynchronously. Because an echo server needs to respond only when data is received, the accept callback function sets up the receive callback as follows.

```
/* set the receive callback for this connection */
tcp_recv(newpcb, recv_callback);
```

When a packet is received, the function recv_callback is called. This function echoes the data it receives back to the sender.

```
/* indicate that the packet has been received */
tcp_recved(tpcb, p->len);

/* echo back the payload */
err = tcp_write(tpcb, p->payload, p->len, 1);
```

Although the RAW API is more complex than the socket API, it provides a higher throughput because it does not have a high overhead.

## Web Server

A simple web server implementation is provided as a reference for TCP-based applications. The web server implements only a subset of the HTTP 1.1 protocol. Such a web server can be used to control or monitor an embedded platform through a browser. The web server demonstrates these features.

- Accessing files residing on a memory file system through HTTP GET commands
- Controlling the LED lights on the development board using the HTTP POST command
- Obtaining status of DIP switches on the development board using the HTTP POST command

The Xilinx memory file system (xilmfs) is used to store a set of files in the memory of the development board. These files can be accessed through an HTTP GET command by pointing a web browser to the IP address of the development board and requesting specific files.

Controlling or monitoring the status of components on the board is done by issuing POST commands to a set of URLs that map to the devices. When the web server receives a POST command to a URL that it recognizes, it calls a specific function to do the work that has been requested. The output of this function is sent back to the web browser in Javascript object notation (JSON) format. The web browser then interprets the data received and updates its display.

The overall structure of the web server is similar to the echo server. There is one main thread that listens on the HTTP port (80) for incoming connections. For every incoming connection, a new thread is spawned that processes the request on that connection.

The HTTP thread first reads the request, identifies whether it is a GET or a POST operation, then performs the appropriate operation. For a GET request, the thread looks for a specific file in the memory file system. If this file is present, it is returned to the web browser initiating the request. If it is not available, an HTTP 404 error code is sent back to the browser.

In socket mode, the HTTP thread is structured as follows.

```
/* read in the request */
if ((read_len = read(sd, recv_buf, RECV_BUF_SIZE)) < 0)
return;

/* respond to request */
generate_response(sd, recv_buf, read_len);
```

Pseudo code for the generate response function is as follows.

```
/* generate and write out an appropriate response for the http request */
    int generate_response(int sd, char *http_req, int http_req_len)
{
        enum http_req_type request_type = decode_http_request(http_req, http_req_len);

        switch(request_type) {
            case HTTP_GET:
                   return do_http_get(sd, http_req, http_req_len);
            case HTTP_POST:
                   return do_http_post(sd, http_req, http_req_len);
            default:
                    return do_404(sd, http_req, http_req_len);
  }
}
```

The RAW mode web server primarily uses callback functions to perform its tasks. When a new connection is accepted, the accept callback function sets up the send and receive callback functions. These are called when sent data has been acknowledged, or when data is received.

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
     /* keep a count of connection # */
     tcp_arg(newpcb, (void*)palloc_arg());
     tcp_recv(newpcb, recv_callback);
     tcp_sent(newpcb, sent_callback);
```

```
        return ERR_OK;
}
```

When a web page is requested, the recv_callback function is called. This function then performs tasks similar to the socket mode function decoding the request and sending the appropriate response.

```
/* acknowledge that we have read the payload */
tcp_recved(tpcb, p->len);

/* read and decipher the request */
/* this function takes care of generating a request, sending it,
* and closing the connection if all data has been sent. If
* not, then it sets up the appropriate arguments to the sent
* callback handler.
*/
generate_response(tpcb, p->payload, p->len);

/* free received packet */
pbuf_free(p);
```

The data transmission is complex. In the socket mode, the application sends data using the lwip_write API. This function blocks if the TCP send buffers are full. However, in RAW mode the application determines how much data can be sent and sends only that much data. Further data can be sent only when space is available in the send buffers. Space becomes available when sent data is acknowledged by the receiver (the client computer). When this occurs, lwIP calls the sent_callback function, indicating that data was sent and there is now space in the send buffers for more data. The sent_callback is structured as follows.

```
err_t sent_callback(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
       int BUFSIZE = 1024, sndbuf, n;
       char buf[BUFSIZE];
       http_arg *a = (http_arg*)arg;

       /* if connection is closed, or there is no data to send */
       if (tpcb->state > ESTABLISHED) {
       return ERR_OK;
       }
       /* read more data out of the file and send it */
       sndbuf = tcp_sndbuf(tpcb);
       if (sndbuf < BUFSIZE)
       return ERR_OK;
       n = mfs_file_read(a->fd, buf, BUFSIZE);
       tcp_write(tpcb, buf, n, 1);

       /* update data structure indicating how many bytes
       * are left to be sent
       */
       a->fsize -= n;
       if (a->fsize == 0) {
              mfs_file_close(a->fd);
              a->fd = 0;
       }
       return ERR_OK;
}
```

Both the sent and receive callbacks are called with an argument that can be set using tcp_arg. For the web server, this argument points to a data structure that maintains a count of the bytes that remain to be sent and the file descriptor that can be used to read this file.

### TFTP Server

TFTP is a UDP-based protocol for sending and receiving files. Because UDP does not guarantee reliable delivery of packets, TFTP implements a protocol to ensure packets are not lost during transfer. See *RFC 1350 - The TFTP Protocol* [Ref 4] for a detailed explanation of the TFTP protocol.

The socket mode TFTP server is very similar to the web server in the application structure. A main thread listens on the TFTP port and spawns a new TFTP thread for each incoming connection request. This TFTP thread implements a subset of the TFTP protocol and supports either read or write requests. At most, only one TFTP data or acknowledge packet can be in flight, which greatly simplifies the implementation of the TFTP protocol. The RAW mode TFTP server is very simplistic and does not handle timeouts. Thus, it is usable only as a point-to-point Ethernet link with zero packet loss. It is provided as a demonstration only.

The TFTP code is very similar to the web server code and hence it is not explained in this application note. The use of UDP allows the minor differences to be understood by examining the source code. See Web Server.

### TCP/UDP RX and TCP/UDP TX Throughput Test

The TCP/UDP transmit and receive throughput test applications are simple applications that determine the maximum TCP transmit and receive throughputs achievable using lwIP, and the Xilinx MAC and PHY combinations. These tests communicate with an open source software called Iperf [Ref 5].

The transmit test measures the transmission throughput from the board running lwIP to the host. In this test, the lwIP application connects to an Iperf server running on a host, and then keeps sending a constant piece of data to the host. Iperf running on the host determines the rate at which data is transmitted and prints it out on the host terminal.

The receive test measures the maximum receive transmission throughput of data at the board. The lwIP application acts as a server. This server accepts connections from any host at a certain port. It receives data sent to it, and silently drops the received data. Iperf (client mode) on the host connects to this server and transmits data to it for as long as needed. At frequent intervals, it computes how much data is transmitted at what throughput and prints this information on the console.

## Using PS GEM through MIO

This section describes how to use the PS Ethernet block GEM3 with the PS PHY through the MIO interface.

# Hardware Design

The PS Ethernet MAC (GEM3) connects the on-board TI PHY through MIO pins using the RGMII interface. The GEM3 block is enabled while generating the hardware system. The GEM3-TI PHY link is shown in Figure 1 with the PS-GEM3 link.

## *Reference Clock Generation*

This design uses the GTX transceiver X0Y4 on the Zynq UltraScale+ MPSoC connected to the SFP cage on the ZCU102 board. The GTX transceiver reference clock (125 MHz differential) is generated from the Si570 jitter attenuator on the ZCU102 board. The clock divider values are adjusted to generate 125 MHz from the Si570 programmable oscillator. The Si570 is programmed over the I2C interface to generate the required clock value of 125 MHz. See the Si570 data sheet [Ref 6] for details on the Si570.

# Software Design

This design uses the common lwIP library (for A53 or R5) for all the GEM on the ZCU102. The lwIP library uses the direct memory access (DMA) controller attached to the GEM controller in the PS. This driver is responsible for several functions, including DMA descriptor rings setup, allocation, and recycling. The interrupt handling is done only for the PS GEM events, because the interrupt status implicitly reflects DMA events.

## *lwIP Library*

An lwIP library (for the Cortex-A53 or Cortex-R5 processor) is provided for this design. The software architecture for PS Ethernet interfaces is shown in Figure 2.

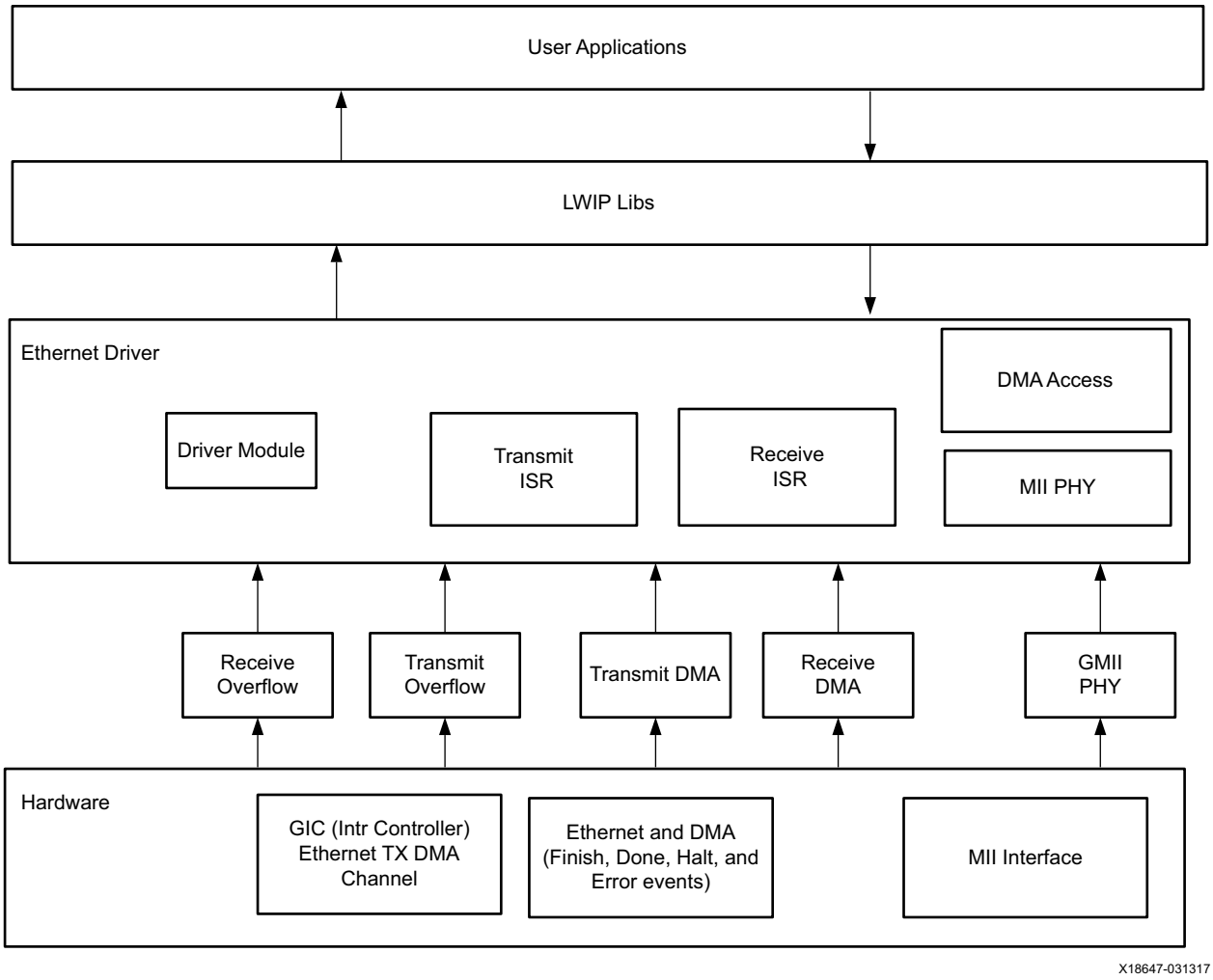


*Figure 2:* **lwIP Library Layers**

## Executing the Reference System

This section describes how to execute the PS Ethernet.

### Host Network Settings

1. Connect the relevant board to an Ethernet port on the host computer using an Ethernet cable.
2. Assign an IP address to the Ethernet interface on the host computer. The software application assigns a default IP address of 10.10.70.3 to the board. This address can be changed in the respective `main.c` files. For this setting, assign an IP address to the host in the same subnet mask as the board, for example 10.10.70.9. See the *PS and Pl based Ethernet in Zynq MPSoC wiki* [Ref 7] for details on the compilation.

# Using PS GEM through EMIO

This section describes how to use the PS Ethernet block GEM0 with the PL PHY through the EMIO interface. The PS Ethernet block is exposed to the PL through the EMIO, GMII, and management data input/output (MDIO) interfaces. The 1G Ethernet PCS/PMA or SGMII core is used as Ethernet physical media in 1000BASE-X mode. High-speed serial transceivers are used to access the small form factor pluggable (SFP) cage on the ZCU102 board. The SFP cage is connected to a standard Ethernet LAN through an SFP-to-RJ45 converter module. To enable the SFP, jumper J16 should be shorted as shown in Figure 5.

## Hardware Design

The PS-PL Ethernet design is shown in Figure 3. The GMII interface connects the PHY and PS GEM through the EMIO pins. The GEM0 block is enabled while generating the hardware system in the Vivado® tools. The PHY address port of the 1G/2.5G Ethernet PCS/PMA or SGMII core can be assigned a fixed value in the range of 1 to 31. See the *1G/2.5G Ethernet PCS/PMA or SGMII v16.0 LogiCORE IP Product Guide* (PG047) [Ref 3] for more information.

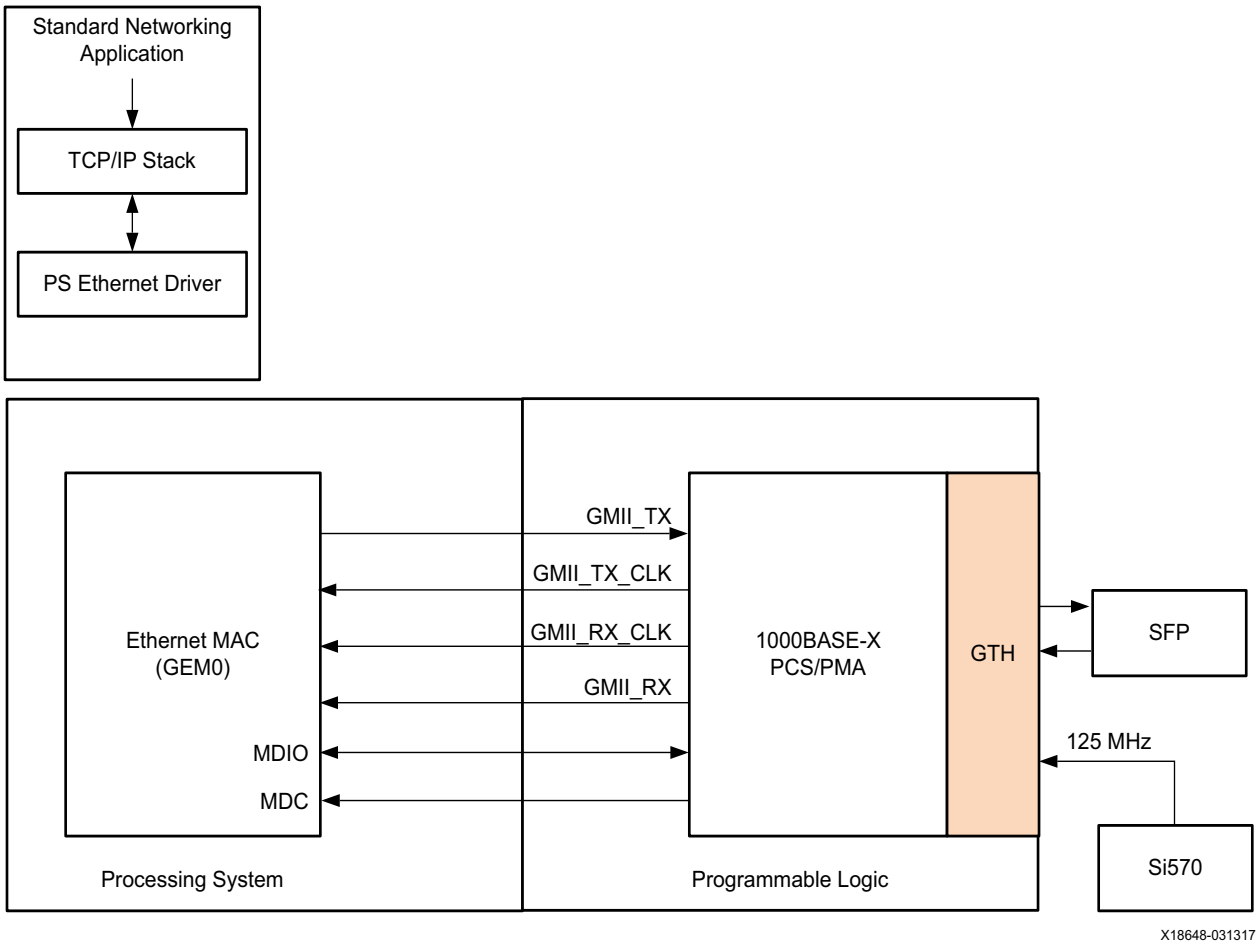


*Figure 3:* **PS-PL Ethernet Design**

### *Reference Clock Generation*

The GTX transceiver X0Y4 on the Zynq UltraScale+ MPSoC is connected to the SFP cage on the ZCU102 board for 1000BASE-X transceivers. The GTX transceiver reference clock (125 MHz differential) is generated from the Si570 jitter attenuator on the ZCU102 board. The clock divider values are adjusted to generate 125 MHz from the Si570 programmable oscillator. The Si570 is programmed over the I2C interface to generate the required clock value. See the Si570 data sheet [Ref 6] for details on the Si570.

To enable GEM0 through the EMIO interface, specific registers must be programmed. This is a part of the PS configuration data used by the Zynq UltraScale+ MPSoC first stage boot loader (FSBL).

To select the EMIO as the source of receive clock, data, and control signals, set the SLCR.GEM0_CLK_CTRL[SRCSEL] bit to `3'b1xx`, where `x` is a don't care (can be either 1 or 0).

## Software Design

This design uses the common lwIP library code for all the GEM on the ZCU102. The lwIP library uses the DMA controller attached to the GEM controller in the PS. This driver is responsible for several functions, including DMA descriptor rings setup, allocation, and recycling. The interrupt handling is done only for the PS GEM events, because the interrupt status implicitly reflects DMA events.

### *lwIP Library*

A lwIP library is provided for this design. Figure 2 shows the software architecture for the PS Ethernet interfaces.

## Executing the Reference System

This section describes how to execute the PS Ethernet.

#### Host Network Settings

1. Connect SFP to RJ-45 converter to SFP 0.
2. Connect the SFP module to an Ethernet port on the host computer through an Ethernet cable.
3. Assign an IP address to the Ethernet interface on the host computer. The software application assigns a default IP address of 10.10.70.3 to the board. This address can be changed in the respective `main.c` files. For this setting, assign an IP address to the host in the same subnet mask as the board, for example 10.10.70.9. See the *PS and Pl based Ethernet in Zynq MPSoC wiki* [Ref 7] for details on the compilation.

# Using PL 1G Ethernet

This section describes the PL implementation of the Ethernet. This design consists of the AXI 1G/2.5G Ethernet subsystem, AXI DMA, and AXI Interconnect IP cores. The AXI 1G/2.5G Ethernet subsystem IP core consists of tri-mode Ethernet MAC (TEMAC) and 1G/2.5G Ethernet PCS/PMA or serial gigabit media independent interface (SGMII) cores. A high performance (HP) port is used in this design for fast access to the PS-DDR memory. The general-purpose slave port can also be used if the HP port is occupied with other peripherals.

## Hardware Design

Ethernet implementation in the PL is shown in Figure 4. The HP port is used for fast data transfers between the PL and the PS DDR4 memory. It connects to the AXI DMA scatter-gather, stream to memory mapped (S2MM), and memory mapped to stream (MM2S) interfaces through the AXI interconnect. This interconnect also performs data-width conversion to connect the 64-bit HP port to the 32-bit interfaces of the AXI DMA. In the AXI DMA, both the scatter-gather option and data realignment engine are enabled for the S2MM and MM2S paths.

The streaming interface of the AXI DMA is connected to the AXI Ethernet subsystem. The AXI Ethernet subsystem has full checksum offloading (CSO) enabled and has FIFO depths of 32K to support jumbo frame transfers.

The AXI Ethernet core implements an Ethernet MAC and supports 1000BASE-X and SGMII PHY interfaces. It connects to the SFP through GTX transceivers through 1000BASE-X/SGMII interfaces.

For the control interface, a general purpose (GP) AXI master port is enabled in the PS. This port connects to the AXI DMA and AXI Ethernet cores.

The 1000BASE-X PHY registers are accessed using the MDIO interface provided through the AXI Ethernet core. The interrupt ports from the AXI DMA and the AXI Ethernet IP cores are connected to the general interrupt controller (GIC) in the PS.

For further details on IP cores, see the *AXI DMA v7.1 LogiCORE IP Product Guide* (PG021) [Ref 8] and *AXI 1G/2.5G Ethernet Subsystem v7.0 Product Guide* (PG138) [Ref 2].

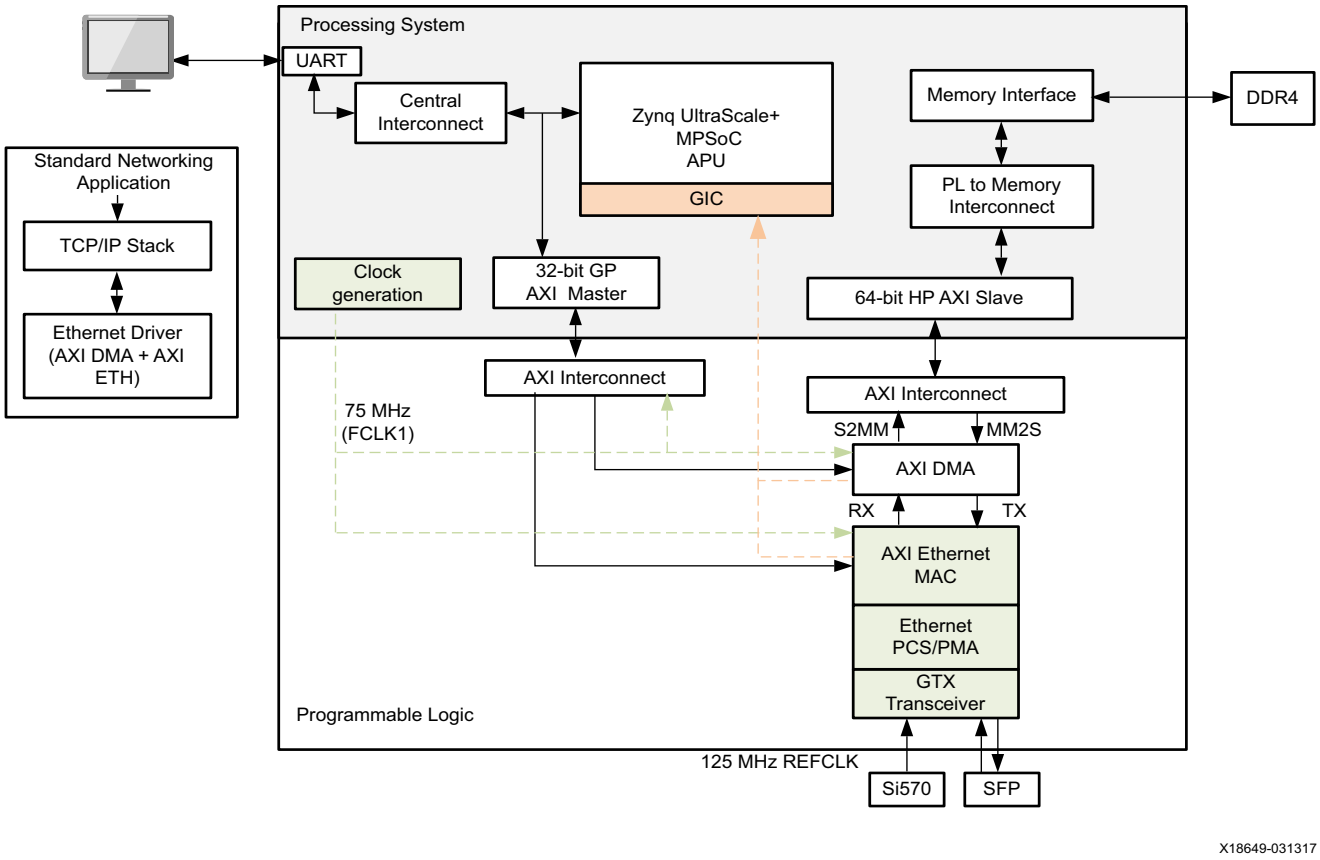


*Figure 4:* **1000BASE-X Ethernet Design**

### *Reference Clock Generation*

The GTX transceiver X0Y4 on the Zynq UltraScale+ MPSoC is connected to the SFP cage on the ZCU102 board for 1000BASE-X transceivers. The GTX transceiver reference clock (125 MHz differential) is generated from the Si570 jitter attenuator on the ZCU102 board. The clock divider values are adjusted to generate 125 MHz from the Si570 programmable oscillator. The Si570 is programmed over the I2C interface to generate the required clock value. See the Si570 data sheet [Ref 6] for details on the Si570.

## Software Design

This section describes the software aspects of the design. The lwIP library facilitates the functionality listed below.

- PL Ethernet MAC accesses
- AXI DMA transfers
- Physical media initialization for 1000BASE-X interface

### *lwIP Library*

A lwIP library is provided for this design. Figure 2 shows the software architecture for the PL Ethernet interfaces.

## Executing the PL Ethernet System

This section describes how to execute the PL Ethernet system.

#### Host Network Settings

1. Connect the relevant board to an Ethernet port on the host computer using an Ethernet cable.
2. Assign an IP address to the Ethernet interface on the host computer. The software application assigns a default IP address of 10.10.70.3 to the board. This address can be changed in the respective `main.c` files. For this setting, assign an IP address to the host in the same subnet mask as the board, for example 10.10.70.9. See the *PS and Pl based Ethernet in Zynq MPSoC wiki* [Ref 7] for details on the compilation.

# Interacting with the Running Software

The socket and RAW mode applications bundle the echo server, web server, TFTP server, and receive and transmit throughput tests examples into a single executable. This output is the same for the Cortex-A53 or Cortex-R5 processor.

## Output from the Application

After the executable is run, the following output appears on the serial port.

```
  -----lwIP RAW Mode Demo Application ------
         Board IP: 10.10.70.3
         Netmask: 255.255.255.0
         Gateway: 10.10.70.1
         auto-negotiated link speed: 1000

        Server   Port   Connect With..
--------------------   ------  --------------------
          echo server   7     $ telnet <board_ip> 7
        rxperf server   5001  $ iperf -c <board ip> -i 5 -t 100
        txperf client   N/A   $ iperf -s -i 5 -t 100 (on host with IP 10.10.70.9)
          tftp server   69    $ tftp -i 192.168.1.10 PUT <source-file>
          http server   80    Point your web browser to http://10.10.70.10
```

For the socket mode application, only the first line changes to indicate that it is the socket mode demonstration application. You can now interact with the application running on the board from the host machine.

## Interacting with the TCP/UDP Receive Throughput Test

To measure receive throughput, connect to the receive iperf application using the iperf client by issuing the iperf -c command with relevant options.

A sample session for TCP is as follows.

```
[root@localhost xhdpssa]# iperf -c 10.10.70.3 -i 5 -t 50 -w 64k
------------------------------------------------------------
Client connecting to 10.10.70.3, TCP port 5001
TCP window size: 128 KByte (WARNING: requested 64.0 KByte)
------------------------------------------------------------
[ 3] local 10.10.70.3  port 43822 connected with 10.10.70.9 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0- 5.0 sec 560 MBytes 939 Mbits/sec
[ 3] 5.0-10.0 sec 562 MBytes 943 Mbits/sec
[ 3] 10.0-15.0 sec 562 MBytes 943 Mbits/sec
[ 3] 15.0-20.0 sec 562 MBytes 943 Mbits/sec
[ 3] 20.0-25.0 sec 562 MBytes 943 Mbits/sec
[ 3] 25.0-30.0 sec 562 MBytes 943 Mbits/sec
[ 3] 30.0-35.0 sec 562 MBytes 943 Mbits/sec
[ 3] 35.0-40.0 sec 562 MBytes 943 Mbits/sec
[ 3] 40.0-45.0 sec 562 MBytes 943 Mbits/sec
[ 3] 45.0-50.0 sec 562 MBytes 943 Mbits/sec
[ 3] 0.0-50.0 sec 5.49 GBytes 942 Mbits/sec
```

A sample session for UDP is as follows.

```
[root@localhost xhdpssa]# iperf -c 10.10.70.3 -i 5 -t 50 -u -b 1G
------------------------------------------------------------
Client connecting to 10.10.70.9, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 3] local 10.10.70.3  port 43822 connected with 10.10.70.9 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-5.0 sec 560 MBytes 955 Mbits/sec
[ 3] 5.0-10.0 sec 562 MBytes 951 Mbits/sec
[ 3] 10.0-15.0 sec 562 MBytes 950 Mbits/sec
[ 3] 15.0-20.0 sec 562 MBytes 952 Mbits/sec
[ 3] 20.0-25.0 sec 562 MBytes 949 Mbits/sec
[ 3] 25.0-30.0 sec 562 MBytes 953 Mbits/sec
[ 3] 30.0-35.0 sec 562 MBytes 952 Mbits/sec
[ 3] 35.0-40.0 sec 562 MBytes 951 Mbits/sec
[ 3] 40.0-45.0 sec 562 MBytes 952 Mbits/sec
[ 3] 45.0-50.0 sec 562 MBytes 954 Mbits/sec
[ 3] 0.0-50.0 sec 5.49 GBytes 952 Mbits/sec
```

***Note:*** To receive maximum performance numbers, use a TCP window size of 8 KB instead of 64 KB.

## Interacting with the TCP/UDP Transmit Throughput Test

To measure the transmit throughput, start the iperf server on the host, and then run the executable on the board. When the executable is run, it attempts to connect to a server at host 10.10.70.9. This address can be changed in the `txperf.c` file.

A sample session host PC for TCP is as follows.

```
[root@localhost xhdpssa]# iperf -s -i 5 -w 64k
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 128 KByte (WARNING: requested 64.0 KByte)
------------------------------------------------------------
[ 4] local 10.10.70.3 port 5001 connected with 10.10.70.9 port 49153
[ ID] Interval Transfer Bandwidth
[ 4] 0.0- 5.0 sec 563 MBytes 944 Mbits/sec
[ 4] 5.0-10.0 sec 566 MBytes 949 Mbits/sec
[ 4] 10.0-15.0 sec 566 MBytes 949 Mbits/sec
[ 4] 15.0-20.0 sec 566 MBytes 949 Mbits/sec
[ 4] 20.0-25.0 sec 566 MBytes 949 Mbits/sec
[ 4] 25.0-30.0 sec 566 MBytes 949 Mbits/sec
[ 4] 30.0-35.0 sec 566 MBytes 949 Mbits/sec
[ 4] 35.0-40.0 sec 566 MBytes 949 Mbits/sec
[ 4] 40.0-45.0 sec 566 MBytes 949 Mbits/sec
[ 4] 45.0-50.0 sec 566 MBytes 949 Mbits/sec
```

A sample session host PC for UDP is as follows.

```
[root@localhost xhdpssa]# iperf -s -i 10 5  -t 100 -u
iperf: ignoring extra argument -- 5
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.5 port 5001 connected with 192.168.1.4 port 42788
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec   958 MBytes   804 Mbits/sec   0.001 ms   58/683527 (0.0085%)
[  3] 10.0-20.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  107/683514 (0.016%)
[  3] 20.0-30.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  231/683499 (0.034%)
[  3] 30.0-40.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  168/683504 (0.025%)
[  3] 40.0-50.0 sec   958 MBytes   803 Mbits/sec   0.001 ms  153/683372 (0.022%)
[  3] 50.0-60.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  144/683404 (0.021%)
[  3] 60.0-70.0 sec   958 MBytes   804 Mbits/sec   0.000 ms   84/683499 (0.012%)
[  3] 70.0-80.0 sec   958 MBytes   804 Mbits/sec   0.002 ms  153/683502 (0.022%)
[  3] 80.0-90.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  155/683512 (0.023%)
[  3] 90.0-100.0 sec   958 MBytes   804 Mbits/sec   0.001 ms  198/683491 (0.029%)
[  3]  0.0-100.0 sec  9.36 GBytes   804 Mbits/sec   0.001 ms 1450/6835471 (0.021%)
```

Press **Ctrl+C** twice to stop the server.

# Hardware and Software Requirements

The following hardware and software is required for testing the designs explained in this application note.

- Standard PC, running the Linux OS
- Ethernet port supporting 1000 Mb/s
- SFP for 1G module on host
- iPerf tool [Ref 5]

- Zynq UltraScale+ MPSoC ZCU102 board with an SFP-to-RJ45 adapter module for testing [Ref 9]
- Vivado tools 2016.4 (IPI Design) [Ref 10]
- PetaLinux 2016.4 XSDK [Ref 11]

The board setup for the 1G interface is shown in Figure 5. Jumper J16 should be set to disable transmission through the SFP. This design was tested with the Cisco GLC-T 1000BASE-X Gigabit Ethernet to optical SFP module.

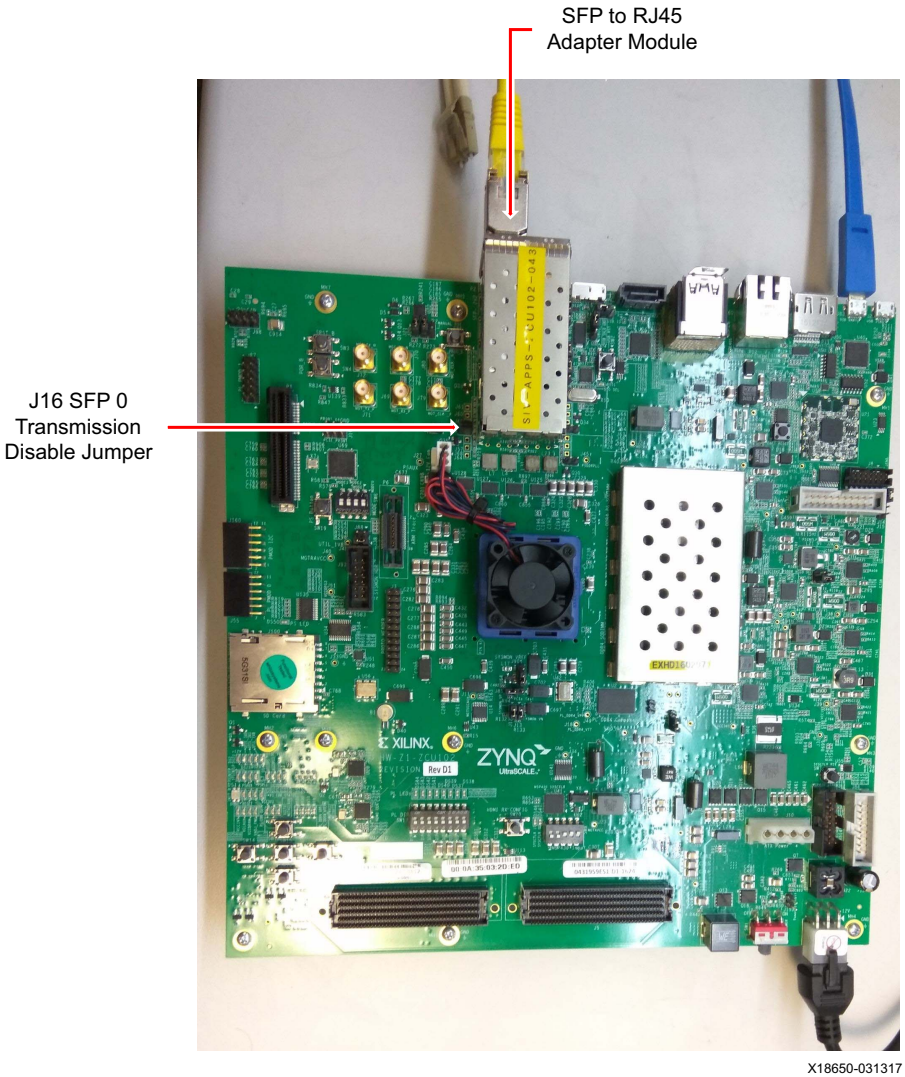


*Figure 5:* **Hardware Setup for 1G Ethernet**

# Conclusion

This application note provides designs for implementing the PS Ethernet through the EMIO/MIO and Ethernet 1G in the PL to support multiple Ethernet links. The performance

benchmarking results for the designs included in this application note can be found in the *PS and Pl based Ethernet in Zynq MPSoC wiki* [Ref 7].

# Reference Design

Download the reference design files for this application note from the Xilinx website.

Table 1 shows the reference design matrix.

*Table 1:* **Reference Design Matrix**

| Parameter | Description |
|---|---|
| **General** | |
| Developers name | Bhargav Shah, Naveen Kumar Gaddipati, Akhilesh Mahajan, and Srini Gaddam |
| Target devices | Zynq UltraScale+ devices |
| Source code provided | Yes |
| Source code format | Verilog, C |
| Design uses code and IP from existing Xilinx application note and reference designs or third party | Yes |
| **Simulation** | |
| Functional simulation performed | No |
| Timing simulation performed | No |
| Test bench used for functional and timing simulations | No |
| Test bench format | N/A |
| Simulator software/version used | N/A |
| SPICE/IBIS simulations | N/A |
| **Implementation** | |
| Synthesis software tools/versions used | Vivado tools 2016.4 |
| Implementation software tools/versions used | Vivado tools 2016.4 |
| Static timing analysis performed | Yes |
| **Hardware Verification** | |
| Hardware verified | Yes |
| Hardware platform used for verification | ZCU102 evaluation board |

# References


1. lwIP (https://en.wikipedia.org/wiki/LwIP)
2. *AXI 1G/2.5G Ethernet Subsystem v7.0 Product Guide* (PG138)
3. *1G/2.5G Ethernet PCS/PMA or SGMII v16.0 LogiCORE IP Product Guide* (PG047)
4. *RFC 1350 - The TFTP Protocol* (http://www.faqs.org/rfcs/rfc1350.html)
5. Iperf (http://sourceforge.net/projects/iperf/)
6. *Si570 Data Sheet* (www.silabs.com/Support%20Documents/TechnicalDocs/Si570.pdf)
7. PS and Pl based Ethernet in Zynq MPSoC
8. *AXI DMA v7.1 LogiCORE IP Product Guide* (PG021)
9. *ZCU102 Evaluation Board User Guide* (UG1182)
10. Xilinx Vivado Design Suite
11. PetaLinux
12. Netperf (www.netperf.org)
13. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
14. *UltraScale Architecture GTH Transceivers User Guide* (UG576)


# Revision History

The following table shows the revision history for this document.

| Date | Version | Revision |
|---|---|---|
| 04/03/2017 | 1.0 | Initial Xilinx release. |

# Please Read: Important Legal Notices

**Automotive Applications Disclaimer**

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.