TCP-UDP-IP STACK 1G v2.0

IP Product Datasheet

PD002

Version 1.0

December 01, 2021

Table of Contents

Introduction

The Gigabit TCP/UDP/IP FPGA IP core implements a high-performance processor-less Internet TCP/UDP/IP protocol stack. The whole Gigabit TCP/IP and/or UDP/IP protocols are implemented within a tiny footprint on the FPGA, which makes the IP core perfectly suitable for either prototyping applications or industrial products.

The UDP/IP stack is designed to achieve a near-line-rate throughput performance. An innovative architecture using a two-level cache for the MAC-IP address routing table (RT) allows one UDP transmitter instance to support up to 8 different destinations without having to consult the main RT each time the destination changes. As for the TCP/IP stack, slow start, congestion avoidance, duplicate ACK detection, fast retransmission and out-of-order packets reassembling techniques are implemented. This offers both the reliability and a high-throughput performance to a TCP/IP connection. As long as the TCP transmitter (TX) buffer covers the packet round trip time (RTT), a near-line-rate throughput can be achieved.

Key Features

- The IP core is compliant to IEEE 802.3 Ethernet packet encapsulation. It supports the IP version 4.
- Various internet protocols are implemented including: Echo Reply, ARP, UDP transmitter, UDP receiver and TCP server/client.
- One instance of the UDP transmitter supports up to 8 different destinations without reducing the global throughput thanks to a two-level ARP cache implementation.
- The UDP TX latency is equal to 21 clock periods (168 ns). A throughput of 968 Mbps is achieved with 1450-byte packets. The UDP RX latency is equal to 54 clock periods (432 ns).
- A TCP segment takes 13 clock periods (104 ns) to go through the whole TX stack, and 42 clock periods (336 ns) for the RX stack. With 1450-byte payload size, the TCP TX achieves a throughput of 958 Mbps. This throughput performance is sustained as long as the TX buffer size is larger than the round trip time. A TCP server can be switched to a TCP client and vice versa in real time.
- TCP Slow Start, Duplicate ACK Detection, Fast Retransmission and Out-of-order packets reassembling techniques are implemented for an optimum TCP transmission.
- Jumbo frame is supported.
- The IP core is designed to interface with the Xilinx Trimode Ethernet MAC. The IP core uses the AXI4 Stream interface for seamless integration.

IP Specification

The block diagram of the IP core is illustrated in Figure 1. The IP core consists of the following principal blocks: MAC Parser, IP Parser, ARP Decoder/Reply, ARP Cache/Request, ICMP/Echo Reply, UDP Transmitter, UDP Receiver and TCP Server/Client.

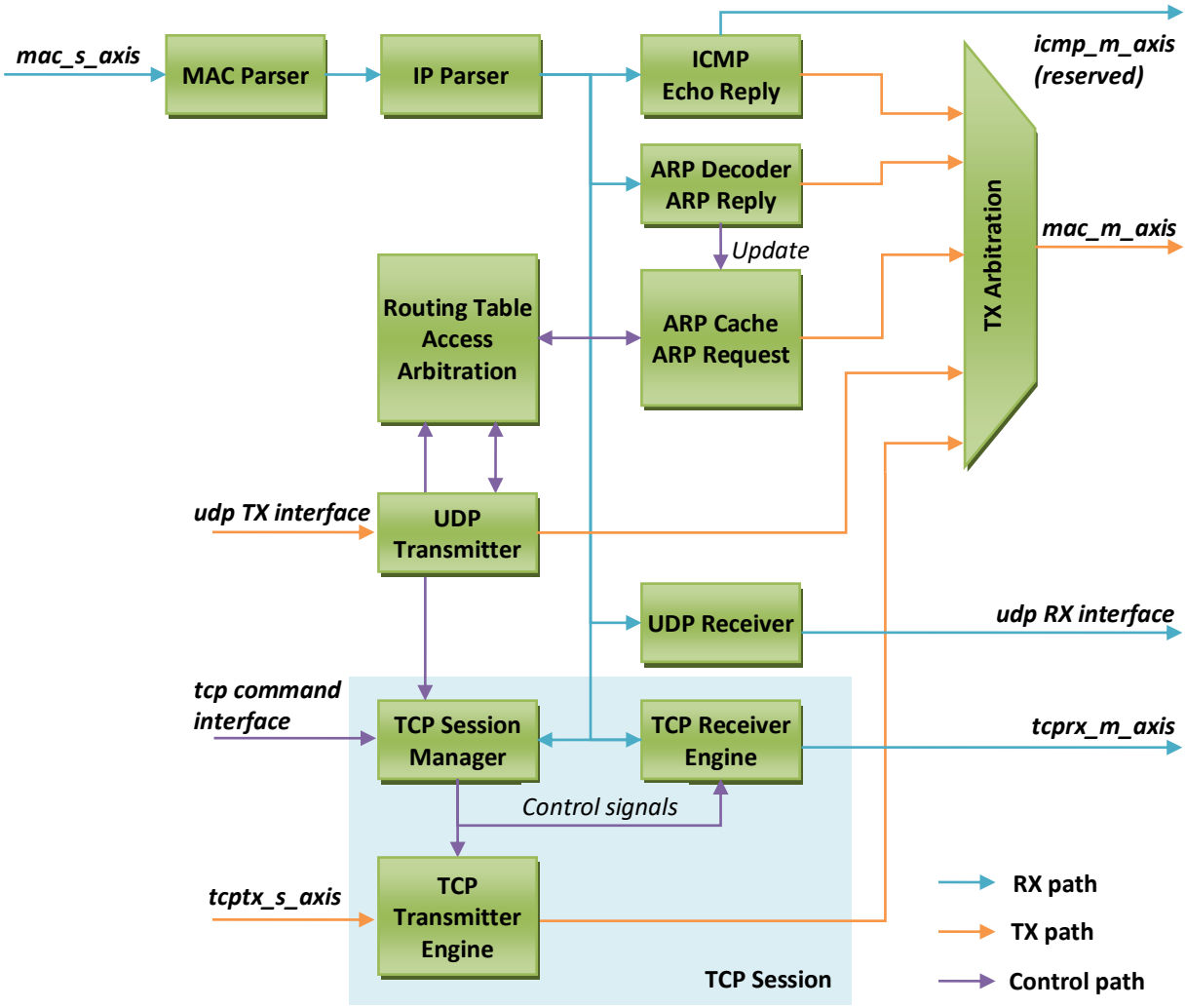

Figure 1- Global architecture block diagram

MAC Parser

The MAC Parser component is responsible for parsing MAC header fields and filtering MAC packets by analyzing the MAC destination address field. Either broadcast packets or packets whose destination address matches the component's MAC address are passed to higher-layer protocols. When promiscuous mode is activated, all packets are accepted.

IP Parser

The IP Parser component carries out the following jobs.

- Verify the IP header CRC.
- Parse all the fields in the IP header.
- Pass payload data to higher-layer protocols.

Packets with a wrong CRC value are discarded. Either broadcast packets or packets whose destination IP address matches the component's IP address are accepted.

When a packet is not an IP packet, it is passed to other protocol parsers such as the ARP decoder module.

In this version, the promiscuous mode is reserved.

ICMP/Echo Reply

ICMP packets are handled by this component. After parsing the type and the code of a message, the component sources the payload to its axis master interface ***icmp_m_axis*** (**Important note**: **this functionality is reserved for future version. For now, this interface should not be used**)**.** Echo request message is however an exception. When an echo request is received by the component, it is passed to the Echo Reply module which then responds to the request.

ARP Decoder/Reply

In IP version 4, neighbor's MAC address is discovered using the ARP protocol. This component listens permanently to ARP packets coming from the MAC Parser. When an ARP request is received, the component initiates an ARP response. When an ARP response or a gratuitous ARP message is received, the component parses the MAC address and the IP address before requesting the ARP Cache component to update the routing table.

At startup or after a reset event, the component automatically sends out a gratuitous ARP message.

ARP Cache/Request

This module establishes and maintains the IP Address - MAC address mapping using the ARP protocol. A routing table is stored in a true dual port RAM. Port A is controlled by a Finite State Machine (FSM) which takes care of RT entry update requests via the ***RT_update*** interface when an ARP response packet has been received. The FSM also refreshes the routing table by periodically sending ARP request packets for "old" entries. "Too old" entries are also removed, leaving place for machines that recently joint the network.

A routing table entry (MAC-IP address pair) can be read by using the ***RT_Request*** interface of the ARP Cache component. Port B of the RAM is controlled by an FSM which takes care of this interface. Upon request, this FSM searches for the entry in the routing table. If the MAC address is found, it responds with an ACK signal. Otherwise, it responds with a NACK signal and then sends out an ARP request packet. If the IP address is not on the same subnet mask, the FSM searches for the gateway's MAC address.

UDP Transmitter

The component has three principal blocks.

- A RAM which stores input UDP packets in ping-pong mode: one packet in, one packet out for minimizing the resource utilization and maximizing the throughput.
- A routing table controller which maintains the IP address - MAC address mapping. It also has its own local ARP cache.
- An input FSM and an output FSM which control the input and output flows.

The routing table controller has its own local ARP cache which maintains up to 8 MAC-IP address pairs. When a transmitted UDP packet has been fully received and stored in the RAM, the input FSM requests the routing table controller for the destination MAC address. If it is found in the local cache, the routing table controller responds an ACK signal to the UDP TX interface. If the requested information is not found in the local cache, the routing table controller sends a request to the main ARP Cache. If an ACK signal along with the destination MAC address is received from the main ARP Cache, the routing table controller registers the destination MAC address, updates its local cache and transfers the ACK signal to the UDP TX interface. If a NACK signal is received, the routing table controller also responds to the UDP TX interface with a NACK signal and the packet is discarded.

When the routing table of the main ARP Cache is large, sometimes the time it takes to find an entry might be long. For a 64-entry routing table, it may take up to 1536 ns to find an entry. In conventional implementation, one UDP TX instance is used for only one destination in order to avoid consulting the main ARP Cache for each packet (in fact we have to consult only once at the beginning). An innovative architecture with a local ARP Cache allows one instance of the UDP Transmitter to support up to 8 different destinations without having to consult the main ARP Cache every time the packet's destination changes. This implementation is extremely advantageous in terms of resource utilization and throughput performance, given that the local Cache read latency is only one clock period.

UDP Receiver

The UDP Receiver component does the following jobs.

- Parse the source port and destination port fields of the UDP header.

- Optionally verify the CRC of the packet.
- Pass payload data to user interface.

At the output of the component, UDP packets can be distinguished by the source port and destination port values. A simple port filter can be implemented to extract the desired datagram.

Available as an option, the UDP Receiver can also verify the UDP checksum of the packet and passes this information to the user interface. This functionality can be useful when users are interested in the data integrity of a transmission even if there is no retransmission mechanism of the UDP protocol. Statistical information concerning the number of packets with wrong CRC values is also available.

TCP Server/Client

The TCP Server/Client module is composed of three principal blocks: a Session Manager, a TX Engine and an RX Engine.

The Session Manager manages the three-way-handshake protocol in order to properly establish and close a TCP connection. This module also resets the TX and the RX engines as long as the connection is not established. Upon a successful connection establishment, this module indicates the initial TX sequence number to the TX Engine and the initial acknowledgement number to the RX Engine before releasing their reset pins. The Session Manager also manages the closing procedure when either the peer initiates a FIN message or the async_tcptx_abort_in signal is triggered by the user.

The TX Engine manages the transmit interface, the sending state machine, Slow Start, Congestion Avoidance, Duplicate ACK detection, Fast Retransmission and Retransmission mechanisms. The sending state machine is carefully designed such that the gap between consecutive transmitted segments is minimized in order to maximize the transmission throughput.

The RX Engine manages segments coming from the peer, updates the peer's ACK number to report to the TX Engine and parses the payload data before sourcing them to the user via the tcprx_m_axis interface. The RX Engine also disposes of a buffer in order to handle out-of-order packet arrival events.

Performance

The IP core’s performance described in this section is obtained with a very specific simulated network condition and with specific IP core’s parameters. When the IP core is used in users’ system, IPCTEK does not guarantee that the same performance will be achieved, given that the users’ network condition and the IP core’s parameters will be changed.

If the latency and/or the throughput performance is critical in users' systems, it is recommended to request to an evaluation version of the IP core and do the test in the real system.

Simulation Setup

In order to evaluate the IP core performance, a test bench with two FPGA IP instances is implemented. One instance serves as a transmitter and the other serves as a receiver.

We place a "switch emulator" module in between the two IP core instances in order to emulate three principal events as follow.

- Switch process delay to emulate different values of the round trip time.
- Packet loss event.
- Out-of-order arrival event.

In the Half Duplex configuration, an additional "Carrier Sense" module is placed in between an IP core instance and a switch emulator.

The simulation synoptic scheme is shown in Figure below.

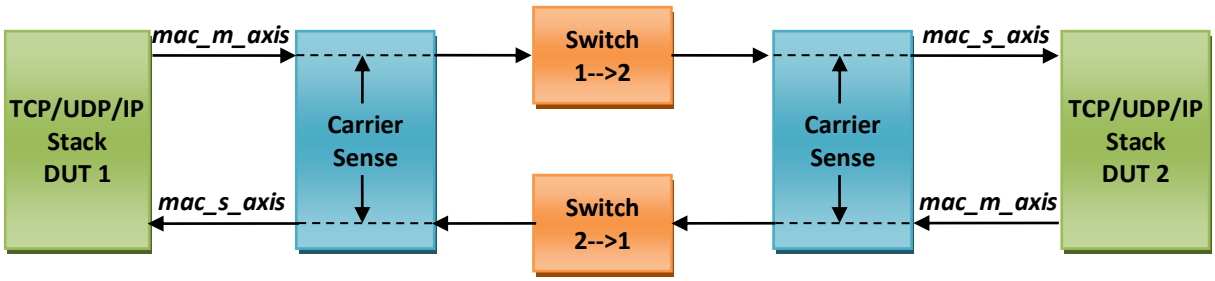

Figure 2- Simulation setup synoptic scheme.

Latency

For TCP TX, TCP RX and UDP TX interfaces, the latency depends on the frame length (datagram length in case of UDP and segment length in case of TCP). For the sake of simplicity, for these interfaces we redefine the latency as follow: the latency of a module is measured by the delay from the last valid data of an input frame to the first valid data of the output frame. In the following paragraphs, we use this definition for the measurement of latency values. If users are interested in the conventional latency (first data of an input frame to the first data of the output frame), the frame length must be taken into account.

UDP TX Latency

As shown in Figure 3, the UDP TX path latency is measured to be equal to 168 ns, which is equivalent to 21 clock periods. For the conventional latency calculation, user must take into account an additional latency due to the UDP frame length. For a frame of size N bytes, the

total latency is equal to (N+21)*8 ns. For example, for a 1450-byte frame, the total latency is equal to (1450 + 21)*8 = 11768 ns.

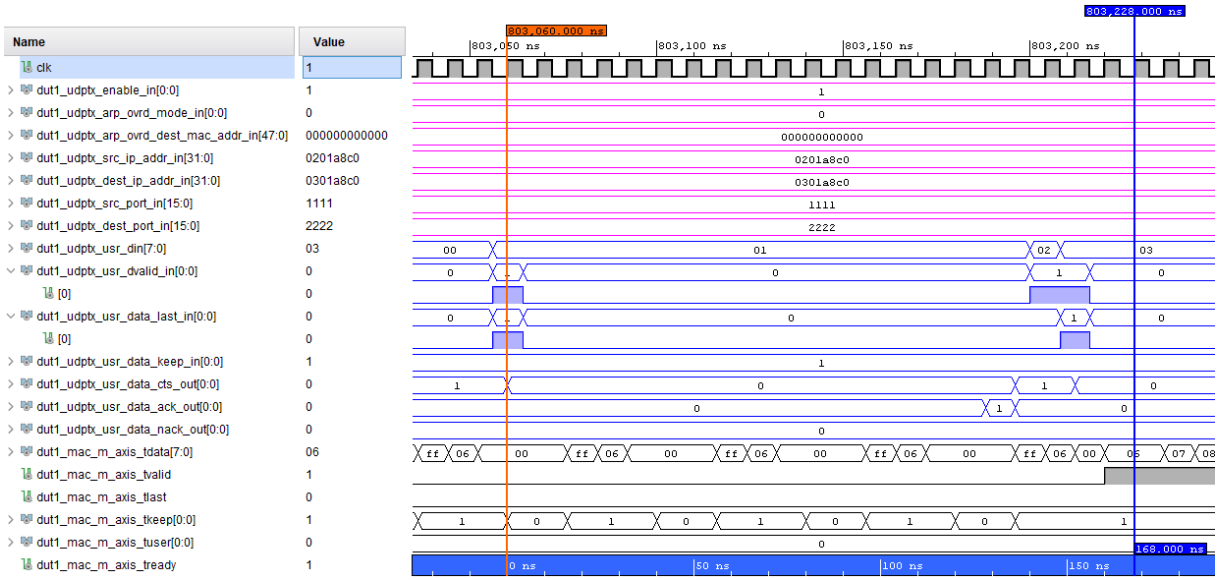

Figure 3- UDP TX latency simulation timing diagram. Magenta waveforms - Datagram parameters. Blue waveforms - TX data interface. Black waveforms - MAC transmit master interface.

UDP RX Latency

Contrarily to the UDP TX, the UDP RX latency does not depend on the UDP datagram length. The conventional latency can be calculated as the delay from the first input valid byte to the first output valid byte as shown in Figure 4. The UDP RX latency is equal to 432 ns, which is equivalent to 54 clock periods.

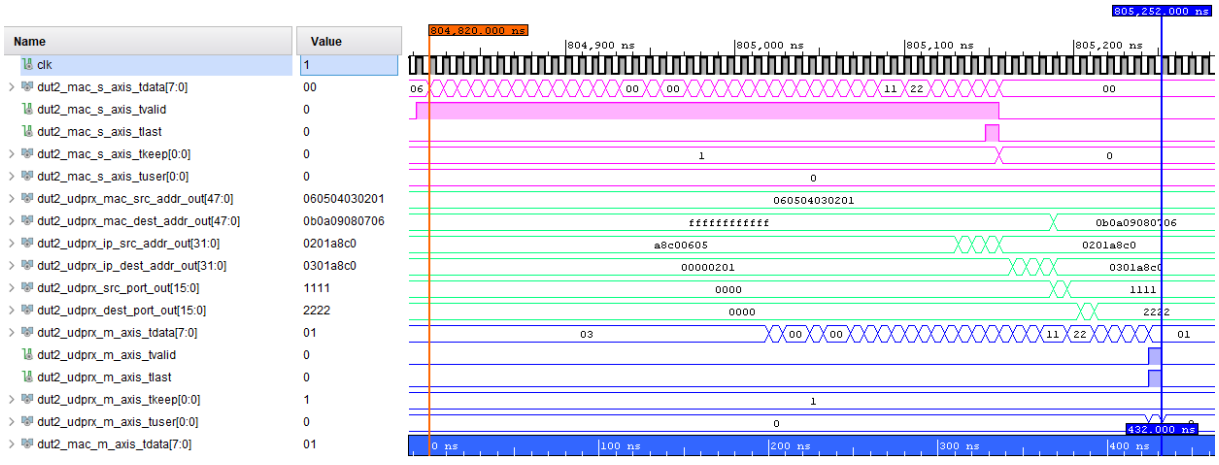

Figure 4- UDP RX latency simulation timing diagram. Magenta waveforms - MAC receive slave interface. Green waveforms - Received datagram parameters. Blue waveforms - Received data master interface.

TCP TX Latency

The TCP TX latency is equal to 104 ns, which is equivalent to 13 clock periods at 125 MHz. For a segment of size N bytes, the total latency is equal to (N+13)*8 ns. For example, for a

1450-byte segment, the total latency is (1450 + 13)*8 = 11704 ns. The Figure below shows the timing diagram of the simulation result.

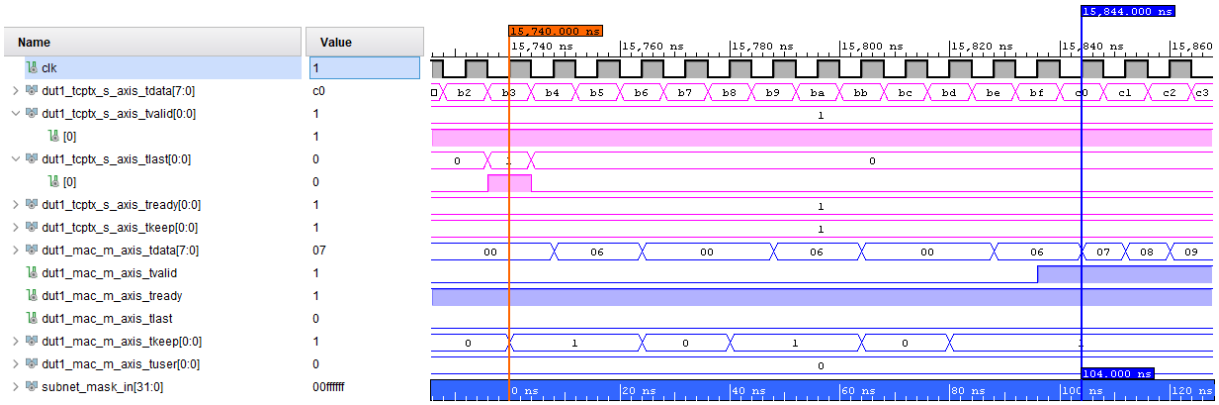

Figure 5- TCP TX latency simulation timing diagram. Magenta waveforms - User transmit slave interface. Blue waveforms - MAC transmit master interface.

In applications where a low latency is important, users should reduce the segment length on the transmitter side.

TCP RX Latency

As shown in the simulation timing diagram, the TCP RX path takes 336 ns (equivalent to 42 clock periods at 125 MHz) to process one segment. For a segment of size N bytes, the total latency is equal to (N+42)*8 ns. For example, for a 1450-byte segment, the total latency is equal to (1450 + 42)*8 = 11936 ns.

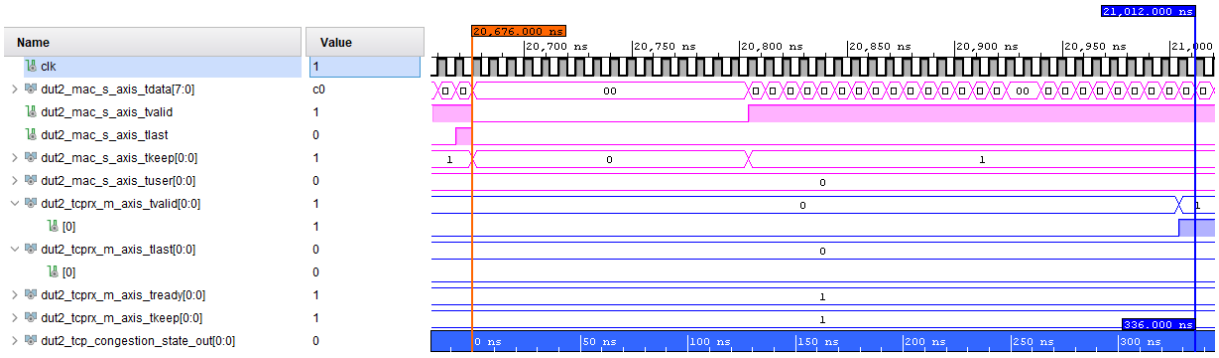

Figure 6- TCP RX latency simulation timing diagram. Magenta waveforms - MAC receive slave interface. Blue waveforms - Received data master interface.

Throughput

UDP Throughput

For 1450-byte frames, the UDP TX (and RX) throughput is measured to be at 968 Mbps.

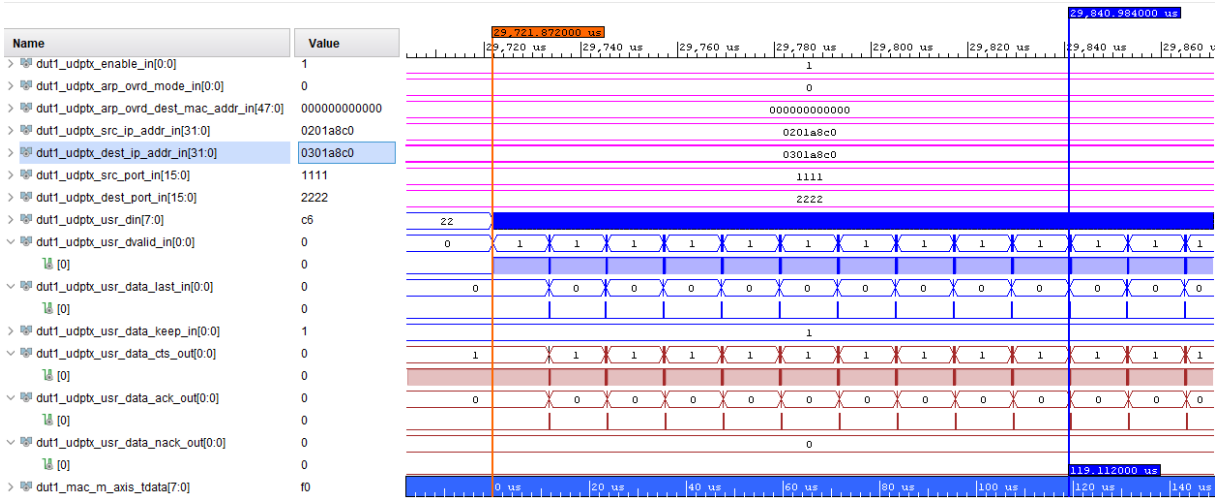

Figure 7- UDP TX throughput simulation timing diagram. Magenta waveforms - datagram parameters. Blue waveforms - transmit data interface. Brown waveforms - Clear-to-send, ACK and NACK signals. Delta cursors show the transmission time of 10 frames.

TCP Throughput

In order to maximize the TCP throughput, the IP core is designed such that segments are transmitted continuously (within the congestion window imposed by Slow Start and the receiver window) to the peer as long as the TX retransmission buffer is not full. When an ACK segment arrives in response to a transmitted segment, the acknowledgement number is verified in another process in parallel with the transmitter process. Hence, as long as the TX retransmission buffer can cover the packet round trip time, the maximal transmission throughput can be achieved.

Figure 8 and Figure 9 show the TCP transmission throughput performance (in simulation) in function of the emulator switch delay for half duplex and full duplex configurations. In both cases, the throughput is sustained at its maximum value as long as the TX buffer still covers the packet round trip time. In point-to-point applications where two FPGAs are directly connected with a RJ45 cable, with a 32-Kbyte TX buffer, these maximum throughput values can be easily obtained. In applications where data are transferred between an FPGA and a CPU, excellent performance can be achieved depending on the CPU performance and that of the TCP/IP stack implemented on the CPU.

Half Duplex

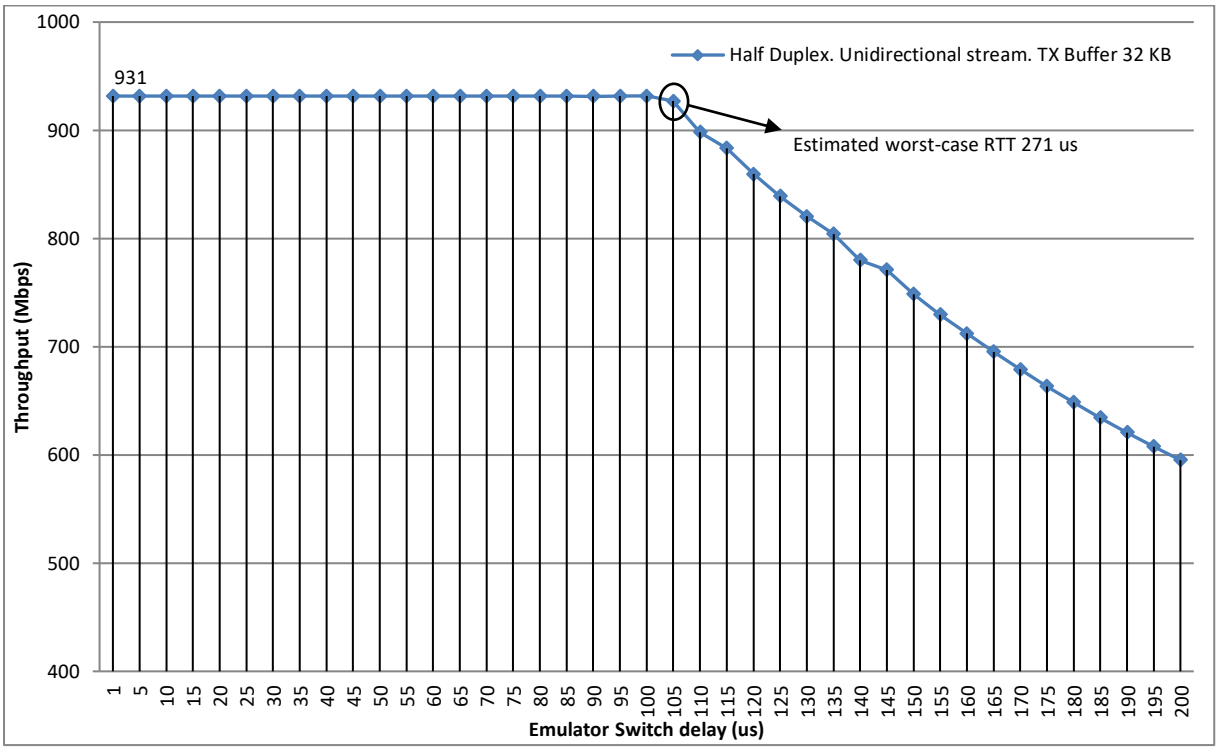

Figure 8- TCPTX transmission throughput in function of emulator switch delay. Half Duplex. Unidirectional stream. TX buffer size 32 KBytes. Segment length 1450 bytes.

In half duplex configuration, 1 Gigabit interface (8 ns for one clock period), the worst-case round trip time can be roughly estimated as follow.

- TX Buffer filling delay: 1450 x 8 = 11600 ns
- Due to carrier sense, if at the output of the MAC there is one arriving ACK packet, we have to wait for 54 x 8 = 432 ns
- Emulator switch TX buffer filling delay: 1502 x 8 = 12016 ns
- Emulator switch process delay (abscissa value at the Figure above), let's take 105000 ns
- When the packet arrives at the destination, if there is one ACK packet coming from the receiver, we have to wait for 54 x 8 = 432 ns
- RX buffer filling delay: 1502 x 8 = 12016 ns

Now we calculate the delay of the return path of the ACK packet.

- Due to carrier sense, at the output of the MAC if there is one arriving packet, we have to wait for 1502 x 8 = 12016 ns
- Emulator switch RX buffer filling delay: 54 x 8 = 432 ns
- Emulator switch process delay: 105000 ns

- When the ACK packet arrives at the transmitter's MAC interface, if there is one packet coming out, we have to wait for 1502 x 8 = 12016 ns
- Transmitter's RX buffer filling delay: 54 x 8 = 432 ns
- Roughly estimated worst-case round trip time is the sum of all these delay values: **271392** ns

It is noted that in practice the round trip time value might be smaller than the estimated value described above because the deferring time due to carrier sense may be smaller depending on whether the two packets arrive at the same time of an interface. With a 32-KByte TX buffer, theoretically the maximum throughput can be achieved as long as the packet round trip time is smaller than 32768 x 8 = **262144** ns.

Full Duplex

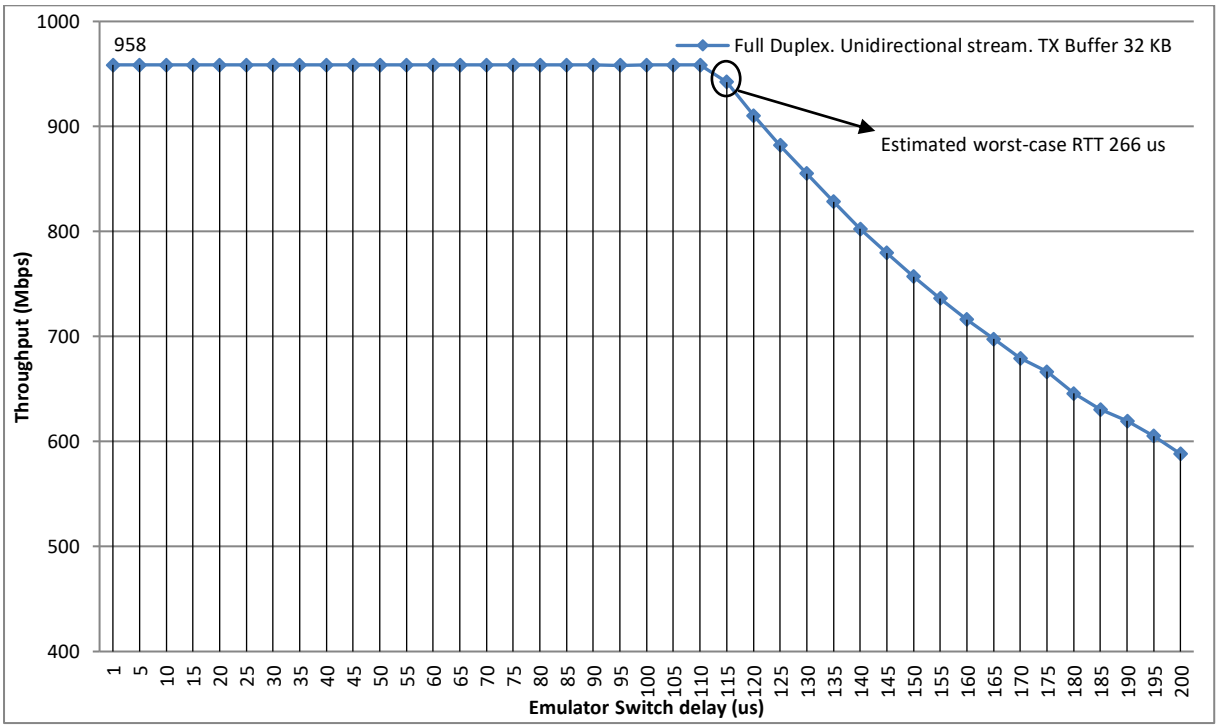

Figure 9- TCPTX transmission throughput in function of emulator switch delay. Full Duplex. Unidirectional stream. TX buffer size 32 KBytes. Segment length 1450 bytes.

For full duplex configuration, 1 Gigabit interface (8 ns for one clock period), the worst-case round trip time can be roughly estimated as follow.

- TX Buffer filling delay: 1450 x 8 = 11600 ns
- Emulator switch TX buffer filling delay: 1502 x 8 = 12016 ns
- Emulator switch process delay (abscissa value at the figure above), let's take 115000 ns
- RX buffer filling delay: 1502 x 8 = 12016 ns

Now we calculate the delay of the return path of the ACK packet.

- Emulator switch RX buffer filling delay: 54 x 8 = 432 ns
- Emulator switch process delay: 115000 ns
- Transmitter's RX buffer filling delay: 54 x 8 = 432 ns

Estimated worst-case round trip time is the sum of all these delay values: **266496** ns

TCP robustness

When evaluating a TCP/IP stack, it is important to see the impact of network disturbance on its performance. In the following paragraphs we will analyze two important problems which arise especially in a network with high congestion: out-of-order packet arrival and packet loss.

Out-Of-Order arrival event

When network traffic load gets high, it is possible that two (or multiple) TCP segments arrive at the destination in reverse order. The segment that was transmitted first may arrive after the segment that was transmitted later. In conventional FPGA TCP/IP stack implementation, out-of-order segments are usually discarded, resulting on a retransmission on the transmitter side. In such implementation strategy, the transmission throughput can be highly degraded.

Available as an option, the IP core can instantiate a buffer and necessary logics to handle out-of-order packet arrival events. Instead of discarding out-of-order packets, they are stored into the buffer to be processed later. Once the missing segments arrive and fill the gap of the reception bytes stream, the buffered segments will be released. This way, the transmitter does not have to retransmit anything, and an excellent transmission throughput can be preserved regardless the out-of-order arrival turbulence.

Figure 10 and Figure 11 illustrate the transmission throughput performance with the presence of out-of-order events, in half duplex and full duplex configuration, respectively. In both configurations, we show the throughput performance of both conventional implementation where out-of-order segments are discarded and the implementation where these segments are wisely handled. In the abscissa we have the out-of-order events frequency (percentage) which means the frequency at which an out-of-order event occurs. For example, a value of 20% means that there is one out-of-order segment out of five transmitted segments. According to simulation results, we can clearly see that when correctly handled, the out-of-order packet arrival disturbance has no significant impact on the transmission throughput. However, in a conventional implementation, the throughput is excessively reduced even with a small number of out-of-order segments.

Half Duplex

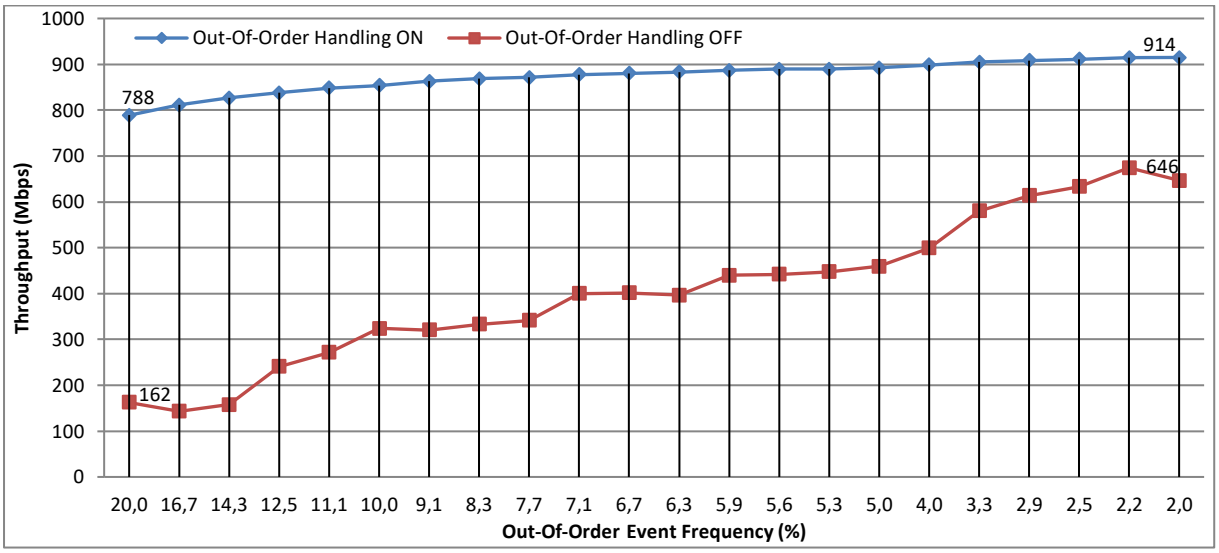

Figure 10- TCP transmission throughput in function of out-of-order arrival event frequency. Half duplex. TX buffer 32 KBytes. Emulator switch delay 10 us.

Full Duplex

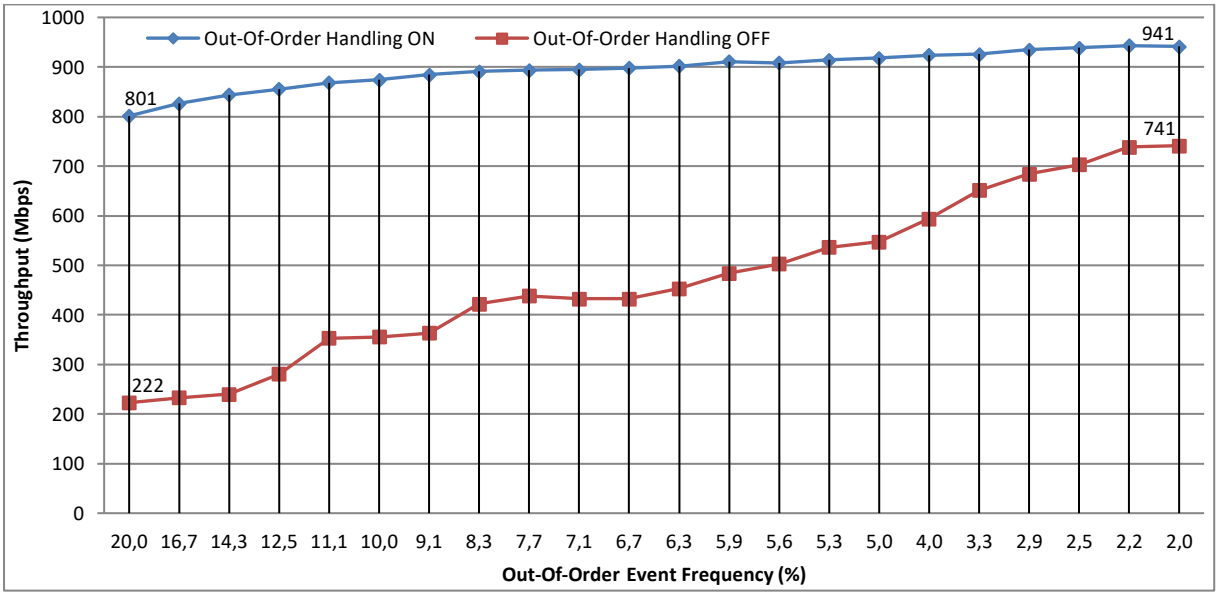

Figure 11- TCP transmission throughput in function of out-of-order arrival event frequency. Full duplex. TX buffer 32 KBytes. Emulator switch delay 10 us.

Packet loss event

In an internet network, it is not rare that a TCP segment did not arrive at the receiver. In such a case, the transmitter has no other options than to retransmit the segment. However, to handle the segments which arrive just after the one that has been lost, we can have two implementation strategies.

The simple and naive implementation strategy is that all these segments are discarded, resulting on a retransmission of both the lost segments and those arriving just after. This implementation is simple at the expense of a significant throughput reduction.

The IPCTEK's FPGA IP core buffers those segments that arrive after the lost one. If the Fast Retransmission technique is implemented at the transmitter, only the lost segment is retransmitted once a duplicate ACK is detected. When the retransmitted segments arrive and fill the gap in the received bytes stream, these segments are released. This way, the transmission throughput can be enhanced.

Figure 12 and Figure 13 show the transmission throughput performance with the presence of packet loss events in half duplex and full duplex configurations, respectively. It is shown that when there is a small amount of packet loss, an excellent throughput is achieved thanks to the out-of-order packet buffer and the Fast Retransmission mechanism, which are all carefully designed and implemented in the IP core.

Half Duplex

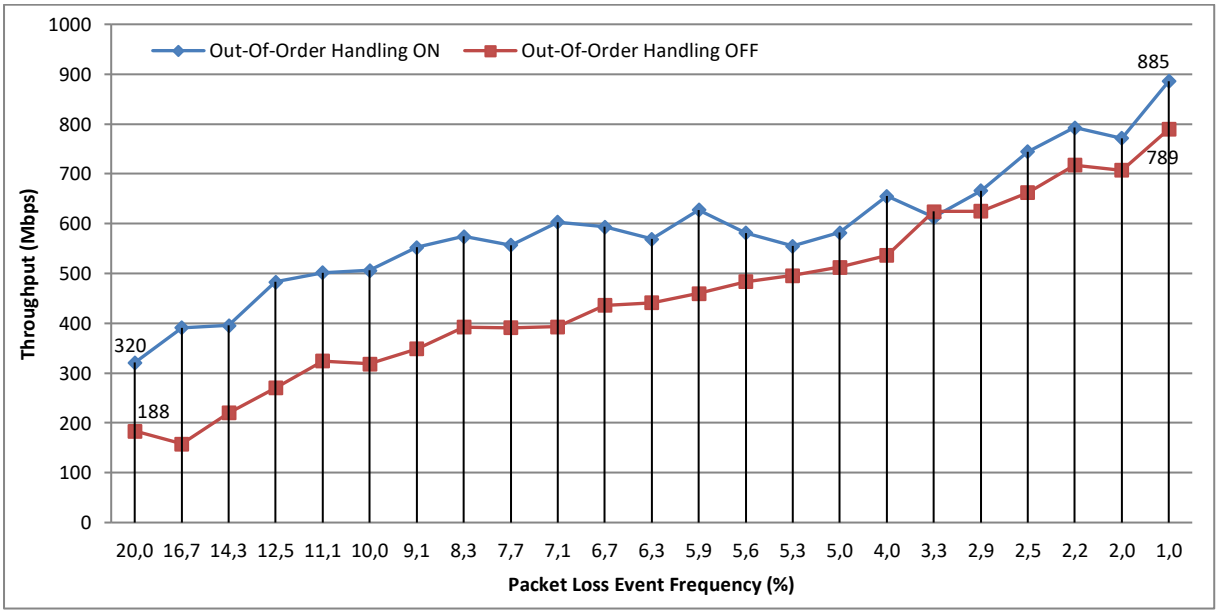

Figure 12- TCP transmission throughput in function of packet loss event frequency. Half duplex. TX buffer 32 KBytes.

Full Duplex

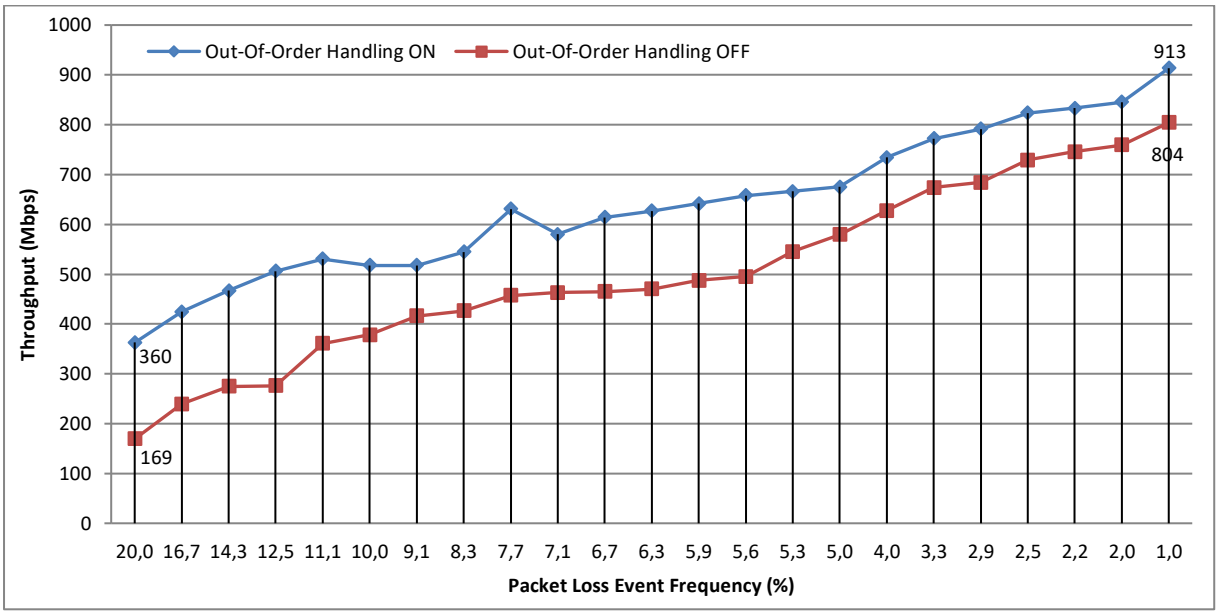

Figure 13- TCP transmission throughput in function of packet loss event frequency. Full duplex. TX buffer 32 KBytes.

Important Note: The out-of-order arrival handling option must only be used when the transmitter does not change the segments' size upon retransmission, which is the case for the IPCTEK's IP core. Hence, when the user uses two instances of the IP core to implement a TCP client and a TCP server, this option can be used. However, if the user is not sure about the implementation of the TCP peer, it is mandatory to disable this out-of-order handling option.

Resource Utilization

Below is the resource utilization after synthesis on the KCU105 board target.

Note: The out-of-order handling buffer is disabled.

Synthesis 1 - UDP only

Parameters	**Configuration 1**	**Configuration 2**
MAC MTU	1500	9000
Number of UDP TX	1	1
Number of UDP RX	1	1
Number of TCP connection	0	0
LUTs utilization	5203	5237
Registers utilization	6421	6458
Block RAM Tile utilization	5	19

Synthesis 2 - UDP and TCP

Parameters	**Configuration 1**	**Configuration 2**
MAC MTU	1500	1500
Number of UDP TX	1	1
Number of UDP RX	1	1
Number of TCP connection	1	2
TCP TX buffer size	16 KBytes	16 KBytes
TCP RX buffer size	16 KBytes	16 KBytes
LUTs utilization	9106	12705
Registers utilization	9883	12394
Block RAM Tile utilization	22	32
Parameters	**Configuration 3**	**Configuration 4**
MAC MTU	1500	9000
Number of UDP TX	1	1
Number of UDP RX	1	1
Number of TCP connection	1	1
TCP TX buffer size	32 KBytes	64 KBytes
TCP RX buffer size	32 KBytes	64 KBytes
LUTs utilization	9099	9256
Registers utilization	9892	10076
Block RAM Tile utilization	30	110

Port Descriptions

This section describes IP ports' functionality.

Port name	Bitwidth	Dir	Description
Common parameters			
mac_tx_clk	1	in	TX path clock. Every interface except the mac_s_axis is synchronous to this clock.
mac_rx_clk	1	in	RX MAC slave interface clock. The mac_s_axis interface is synchronous to this clock.
arst_n	1	in	Asynchronous reset. To reset the IP, this signal must be held low for at least 2 periods of the mac_tx_clk.
promicuous_mode_in	1	in	Promiscuous mode. This signal is reserved for future use.
mac_addr_in	48	in	Component's MAC address. Convention: Read the MAC address from the LSB octet to the MSB octet. E.g. 0xBBAA99887766 stands for conventional MAC address 66:77:88:99:AA:BB
subnet_mask_in	32	in	Subnet mask. Because the IP address is read from LSB to MSB, for 12-bit subnet mask 255.240.0.0 the subnet_mask_in = 0xF0FF
ip_addr_in	32	in	Component's IP address. Read the IP address from the LSB octet to the MSB octet. For example, 0x0201A8C0 stands for 192.168.1.2
gateway_ip_addr_in	32	in	Gateway's IP address. Read the IP address from the LSB octet to the MSB octet.
echo_rep_enable_in	1	in	Echo reply enable signal. This signal must be asserted after the component's IP address has been allocated.
rt_old_entry_microsec_threshold_in	64	in	Timestamp threshold after which a routing table entry need to be refreshed. Unit: microsecond.
rt_entry_to_rm_microsec_threshold_in	64	in	Timestamp threshold after which a routing table entry need to be removed (too old entry). Unity: microsecond.
rt_refresh_timeout_in	64	in	Routing table refreshment timeout: the period at which the routing table is refreshed. Unit: microsecond.
ARP_resp_wait_timeout_in	64	in	ARP response waiting timeout. When an ARP request packet is sent in order to refresh an old entry in the routing table, the Update FSM goes to READY_FOR_UPDATE state waiting for an ARP response message. This signal gives the timeout duration of this state. A value larger than the max Round Trip Time should be assigned to this port. At the timeout event, the FSM continues to refresh other entries in the routing table. Unit: microsecond.
RX Slave axis interface with the Ethernet MAC			
mac_s_axis_tdata	8	in	MAC RX data.
mac_s_axis_tvalid	1	in	MAC RX data valid.
mac_s_axis_tlast	1	in	MAC RX last byte in a frame.
mac_s_axis_tkeep	1	in	MAC RX data valid mask of the data word. For 1G stack this signal is always equal to '1'.
mac_s_axis_tuser	1	in	MAC RX data error, indicating an error event of a packet. This signal is valid only when both mac_s_axis_tvalid and mac_s_axis_tlast signals are asserted. '1' Packet has error. '0' Packet without error.
TX Master axis interface with the Ethernet MAC			

mac_m_axis_tdata	8	out	MAC TX data.
mac_m_axis_tvalid	1	out	MAC TX data valid.
mac_m_axis_tready	1	in	MAC TX data ready given by the Ethernet MAC.
mac_m_axis_tlast	1	out	Indicate the last byte in a TX MAC frame.
mac_m_axis_tkeep	1	out	MAC TX valid mask of the data field. For 1G stack this signal is always equal to '1'.
mac_m_axis_tuser	1	out	Underrun indication. This signal is asserted when an underrun event occurs.
ICMP message interface (reserved)			
icmp_mac_src_addr_out	48	out	MAC address of the sender of this packet.
icmp_mac_dest_addr_out	48	out	MAC address of the destination of this packet.
icmp_ip_src_addr_out	32	out	IP source address of the output datagram.
icmp_ip_dest_addr_out	32	out	IP destination address. Normally this is our IP address.
icmp_type_out	8	out	ICMP type. This signal is valid from the first until the last tvalid (a whole packet)
icmp_code_out	8	out	ICMP code. This signal is valid from the first until the last tvalid (during a whole packet)
icmp_m_axis_tdata	8	out	ICMP data. IP header and 64 bits of original packet.
icmp_m_axis_tvalid	1	out	ICMP data valid.
icmp_m_axis_tlast	1	out	ICMP tlast signal.
icmp_m_axis_tkeep	1	out	Always equal to '1'.
icmp_m_axis_tuser	1	out	'1' → Good frame (CRC check passed). '0' → Bad frame (CRC checked not passed or propagated error from the MAC). This signal is valid when both icmp_m_axis_tvalid and icmp_m_axis_tlast signals are asserted.
UDP TX interface			
udptx_src_ip_addr_in	32xNB_TXs	in	UDP source IP address. Read the IP address from the LSB octet to the MSB octet. Bitwidth of 32 bits for each TX instance. NB_TXs is the number of TX instances.
udptx_dest_ip_addr_in	32xNB_TXs	in	UDP destination IP address. Read the IP address from the LSB octet to the MSB octet. Bitwidth of 32 bits for each TX instance.
udptx_src_port_in	16xNB_TXs	in	UDP source port. Bitwidth of 16 bits for each TX instance.
udptx_dest_port_in	16xNB_TXs	in	UDP destination port. Bitwidth of 16 bits for each TX instance.
udptx_arp_ovrd_mode_in	NB_TXs	in	ARP override mode. In this mode the module is forced to use the user-defined destination MAC address without requesting the routing table. This can be used for example by the DHCP client component during the address acquisition process. 1 bit for each TX instance.
udptx_arp_ovrd_dest_mac_addr_in	48xNB_TXs	in	User-defined destination MAC address used in the ARP override mode. Bitwidth of 48 bits for each TX instance. The MAC address is read from the LSB octet to the MSB octet.
udptx_enable_in	NB_TXs	in	UDP TX Enable signal. 1 bit for each TX instance. '1' → Enabled '0' → Disabled
UDP TX data interface			
udptx_usr_din	8xNB_TXs	in	TX data. Bitwidth of 8 bits for each TX instance.
udptx_usr_dvalid_in	NB_TXs	in	TX data valid. 1 bit for each TX instance.
udptx_usr_data_last_in	NB_TXs	in	Indicate the last byte of an UDP packet. 1 bit for each TX.
udptx_usr_data_keep_in	NB_TXs	in	Data byte mask. Always equal to '1'. 1 bit for each TX.
udptx_usr_data_cts_out	NB_TXs	out	Clear to send signal. Ready for the reception of data. When

			this signal is asserted, the UDP TX FIFO has enough free room to receive a whole packet whose size is limited by the MAC MTU value, after excluding the IP and UDP headers.
udptx_usr_data_ack_out	NB_TXs	out	If a packet is successfully sent to the Ethernet MAC, the module responds with an ack signal. In this case this signal is asserted '1' for one clock period. 1 bit for each TX instance.
udptx_usr_data_nack_out	NB_TXs	out	If a packet is discarded because the destination MAC address is unknown by the routing table, the UDP TX module responds with a nack signal. In this case this signal is asserted '1' for one clock period. 1 bit for each TX instance.
UDP Receiver interface			
udp_checksum_cal_bypass_in	1	in	Bypass the checksum calculation at the receiver. '1' → Bypass the checksum calculation '0' → The checksum is calculated by the receiver
udprx_mac_src_addr_out	48	out	MAC address of the sender of this packet. Read the MAC address from the LSB octet to the MSB octet.
udprx_mac_dest_addr_out	48	out	MAC address of the destination of this packet. Read the MAC address from the LSB octet to the MSB octet.
udprx_ip_src_addr_out	32	out	IP address of the packet's sender. Read from LSB to MSB.
udprx_ip_dest_addr_out	32	out	IP address of the packet's receiver. Read from LSB to MSB.
udprx_src_port_out	16	out	UDP source port of this packet.
udprx_dest_port_out	16	out	UDP destination port of this packet.
			These 6 signals above are valid from the first tvalid to the last tvalid of a received frame (they stay valid for a whole UDP frame)
udprx_m_axis_tdata	8	out	UDP packet payload data.
udprx_m_axis_tvalid	1	out	UDP payload data valid.
udprx_m_axis_tlast	1	out	Indicate the last byte in an UDP payload packet.
udprx_m_axis_tkeep	1	out	Data valid byte mask. Always equal to '1' in this 1G stack.
udprx_m_axis_tuser	1	out	This signal is asserted along with the udprx_m_axis_tvalid and the udprx_m_axis_tlast signals if the UDP checksum is present and its value is correct. This is useful for data integrity verification. '1' → good CRC. '0' → bad CRC.
udprx_nb_pck_err_out	32	out	Number of packets with propagated error. This error is propagated from the MAC. In this version it is not used.
udprx_nb_pck_cs_err_out	32	out	Number of packets with checksum error.
udprx_nb_pck_err_total_out	32	out	Total number of packets with error. In this version this signal is reserved.
TCP Connection parameters interface			
tcprx_out_of_order_handle_enable_in	1	in	Out-of-order arrival event handling enable. '1' → Handle out-of-order arrival events '0' → Do not handle out-of-order arrival events
tcptx_MSS_in	16	in	Maximum segment size in number of bytes. This is used during connection handshake and for the TX segment descriptor construction.
tcptx_use_user_defined_reTX_value_in	1	in	Indicate whether the user-defined or the estimated round trip time value is used for the retransmission timer. '1' → User-defined value is used '0' → Estimated value is used
tcptx_user_defined_reTX_timeout_usec_in	32	in	User-defined retransmission timeout in microseconds.
tcptx_init_seq_num_in	32	in	Our initial sequence number. Used in the SYNC message in

			three-way-handshake TCP connection protocol.
tcptx_srv_mode_in	NB_TCP_CNTNS	in	Server/Client mode selection. '1' → Server mode. '0' → Client mode. NB_TCP_CNTNS is the number of connection instances. 1 bit for each instance.
tcptx_tcp_src_port_in	16xNB_TCP_CNTNS	in	TCP source port (our port). Bitwidth of 16 bits for each connection instance.
tcptx_tcp_dest_port_in	16xNB_TCP_CNTNS	in	TCP destination port (peer's port). This is only used when we are in client mode to indicate the server port to connect to. Bitwidth of 16 bits for each connection instance.
tcptx_server_ip_addr_in	32xNB_TCP_CNTNS	in	Server's (peer's) IP address. This is only used when we are in client mode to indicate the server's IP address. Bitwidth of 32 bits for each connection instance.
TCP Connection commands interface			
async_tcptx_open_in	NB_TCP_CNTNS	in	Asynchronous trigger to open a session. In the server mode, this command is used to open a server at port tcptx_tcp_src_port_in and listen to a connection request. In the client mode, this is used to send a connection request to the server. 1 bit for each connection instance. Required duration: at least 2 periods of mac_tx_clk.
tcptx_open_rdy_out	NB_TCP_CNTNS	out	This signal indicates whether the module is ready for the open command. 1 bit for each connection instance.
async_tcptx_abort_in	NB_TCP_CNTNS	in	This command is used to abort the actual connection. 1 bit for each connection instance.
tcptx_state_out	4xNB_TCP_CNTNS	out	Report the state of the connection state machine. 4 bits for each connection instance. "0000" => CLOSE "0001" => RT_REQ "0010" => WAIT_RT_RESP "0011" => LISTEN "0100" => SYNC_RCVD "0101" => SYNC_SENT "0110" => ESTAB "0111" => FIN_WAIT1 "1000" => CLOSE_WAIT "1001" => FIN_WAIT2 "1010" => LAST_ACK "1011" => TIME_WAIT "1100" => IP_ID_REQ "1101" => SEND "1110" => TX_CLOSE
tcptx_cntn_established_out	NB_TCP_CNTNS	out	Indicate whether the connection is established. 1 bit for each connection instance. '1' → Connection has been established. '0' → Not connected.
cntn_id_mac_src_addr_out	48*NB_TCP_CNTNS	out	After the connection establishment, this port reports the peer's MAC address. 48 bits for each connection. The MAC address is read from the LSB octet to the MSB octet.
cntn_id_ip_src_addr_out	32*NB_TCP_CNTNS	out	After the connection establishment, this port reports the peer's IP address. 32 bits for each connection. The IP address is read from the LSB octet to the MSB octet.
cntn_id_tcp_src_port_out	16*NB_TCP_CNTNS	out	After the connection establishment, this port reports the peer's port. 16 bits for each connection.

cntn_id_tcp_dest_port_out	16*NB_TCP_CNTNS	out	After the connection establishment, this port reports the peer's destination port, which is logically equal to our port. 16 bits for each connection.
TCP TX axis data interface			
tcptx_s_axis_tdata	8*NB_TCP_CNTNS	in	TCP TX data. Bitwidth of 8 bits for each connection instance.
tcptx_s_axis_tvalid	NB_TCP_CNTNS	in	TCP TX data valid. 1 bit for each connection instance.
tcptx_s_axis_tready	NB_TCP_CNTNS	out	TCP TX ready to receive data. Note that a byte is only consumed by the module when both tvalid and tready signals are asserted. 1 bit for each connection instance.
tcptx_s_axis_tlast	NB_TCP_CNTNS	in	User can assert tlast signal to indicate an end-of-segment event. When the module receives tlast, it constructs right away a TX descriptor and tries to send the segment right after. When tlast is not asserted during TX, the module consumes tcptx_MSS_in bytes before constructing a descriptor to send the segment. When tlast is asserted, the segment size can be smaller than the tcptx_MSS_in value. This mode is useful in situations where user wants to send small segments to the other end with a minimum latency. 1 bit for each connection instance.
tcptx_s_axis_tkeep	NB_TCP_CNTNS	in	In this 1G stack, this signal must be always equal to '1'. 1 bit for each connection instance.
TCP RX axis data interface			
tcprx_m_axis_tdata	8*NB_TCP_CNTNS	out	TCP RX data. Bitwidth of 8 bits for each connection instance.
tcprx_m_axis_tvalid	NB_TCP_CNTNS	out	TCP RX data valid. 1 bit for each connection instance.
tcprx_m_axis_tlast	NB_TCP_CNTNS	out	TCP RX tlast signal indicating the last byte of a segment. 1 bit for each connection instance.
tcprx_m_axis_tready	NB_TCP_CNTNS	in	TCP RX ready to receive data. The TCP RX module supports back pressure (thanks to receiver FIFOs) when user logic is not ready to consume the RX data. For an optimum TCP transmission throughput, this signal should always be asserted. 1 bit for each connection instance.
tcprx_m_axis_tkeep	NB_TCP_CNTNS	out	Always equal to '1'. 1 bit for each connection instance.
TCP debug interface			
tcp_congestion_state_out	NB_TCP_CNTNS	out	Congestion window state machine. '0' → Slow Start. '1' → Congestion Avoidance. 1 bit for each connection instance.
tcp_congestion_wdn_out	16*NB_TCP_CNTNS	out	Congestion window. Bitwidth of 16 bits for each connection instance.
tcp_TX_engine_nb_bytes_sent_out	64*NB_TCP_CNTNS	out	Number of bytes sent since the last reset. Bitwidth of 64 bits for each connection instance.
tcp_RTT_usec_out	32*NB_TCP_CNTNS	out	Estimated RTT value for the TX path. Bitwidth of 32 bits for each connection instance. Unit: microsecond.
tcp_TX_engine_nb_bytes_sent_out	64*NB_TCP_CNTNS	out	Number of bytes received since the last reset. Bitwidth of 64 bits for each connection instance.
Miscellaneous			
ipc_ip_version	16	out	IP version. Should read 0x200.
ipc_ip_demo	1	out	'1' → Demo design, the IP stops after 10 minutes. '0' → Released IP.
ipc_ip_mtu	16	out	Reflect the MAC MTU parameter.
ipc_ip_nb_tcp_ssn	16	out	Reflect the number of instantiated TCP connections.

Design Guidelines

This section details important guidelines in order to successfully integrate the IP core into user FPGA design.

Clocking

In order to achieve a simple and efficient design, internally the design is fully synchronous with the clock ***mac_tx_clk.*** However, the IP core also supports asynchronous TX and RX paths where the TX clock may be asynchronous to the RX clock. In such an asynchronous configuration, a FIFO is placed just after the ***mac_s_axis*** interface in order to change the clock domain ***mac_rx_clk*** to the clock domain ***mac_tx_clk***. After the FIFO, all the logics are fully synchronous to only one clock domain, which is the ***mac_tx_clk.*** It is important to note that both transmit path and receive path are synchronous to this clock.

In the user design, the ***mac_tx_clk*** is simply connected to the transmit path clock and the ***mac_rx_clk*** is connected to the receive path clock. In synchronous design (usually the clock is given by the Ethernet PHY), just connect these two clocks to the same clock source. As for the user's logics, everything is synchronous to the ***mac_tx_clk*** due to the FIFO described above.

Reset

After receiving stable clock sources, the IP need to be resetted for a correct functioning. To do so, the IP port ***arst_n*** need to be held low for at least 2 periods of the ***mac_tx_clk***. It is noted that this reset port is asynchronous hence the reset signal does not need to be synchronous to the clock signal. After receiving the reset signal, the IP core generates necessary different synchronous reset signals to reset different logics.

Axis basic handshake

The MAC slave interface (receiver), MAC master interface (transmitter), TCP TX and TCP RX interfaces are compliant to the AXI Stream (AXIS) interface. During an AXIS transaction, we have a master which sources the data and a slave which sinks the data. The function of a signal in an AXIS interface can be decoded according to its suffix.

- _tdata: this signal contains the data for the transaction.
- _tvalid: this signal indicates that the data is valid.
- _tready: this signal indicates that the slave is ready to sink the data.
- _tlast: this signal indicates that this data is the last in a frame.

- _tkeep: when the bytewidth of the data is larger than 1, the tkeep signal indicates which bytes in the data are valid. Each bit in the tkeep signal corresponds to one byte in the data signal. The LSB bit corresponds to the LSB byte and so on.
- _tuser: this signal contains additional information concerning the frame.

In order to successfully integrate a component with an AXIS interface into your design, it is worth taking care of several basic rules.

- The data is only successfully consumed by the slave when both tvalid and tready signals are asserted.
- The master should never wait for the slave's tready signal to be asserted before asserting the tvalid signal. This may result in a deadlock situation in some cases.
- User should assert the tlast signal at the end of a frame, especially when handshaking with the TCP TX interface. When the transmitted segment is small and the tlast signal is missing, the segment risks not to be sent right away to the other end. This is because when the tlast signal is missing, the TCP TX module waits for maximum-segment-size bytes before constructing a descriptor to send the segment.

Figure below shows the basic handshake of an AXIS interface.

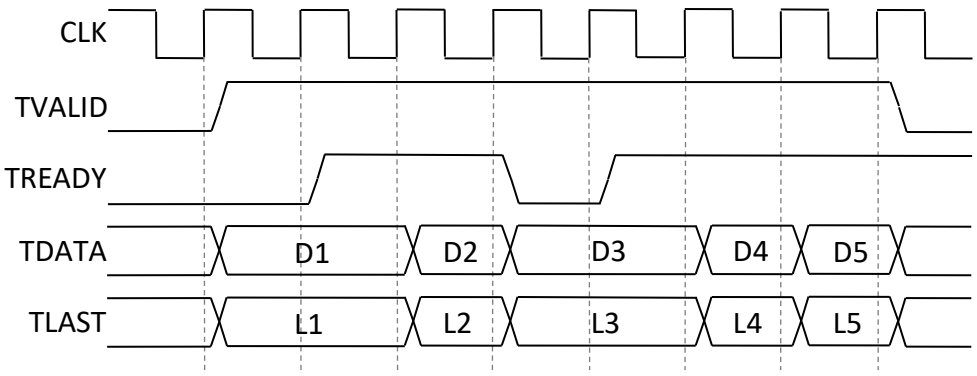

Figure 14- AXI-Stream basic handshake.

UDP TX Interface

The UDP TX engine works on a per packet basis. The basic state machine that can be used to interface with the UDP TX engine can be described as follows.

- **Step 1**: The source and destination IP addresses, the UDP source and destination ports are placed at the input of the IP core.
- **Step 2**: Wait until the signal "clear to send" (***udptx_usr_data_cts_out***) is asserted.
- **Step 3**: Place the data to the transmit interface, do not forget to assert the equivalent tlast signal (***udptx_usr_data_last_in***) to delimit the end of a packet.

- **Step 4**: Wait for either ACK signal (***udptx_usr_data_ack_out***) or Not-ACK signal (***udptx_usr_data_nack_out***) to be asserted.
- The transmission of one UDP packet is finished. Come back to **step 1** to continue the next packet.

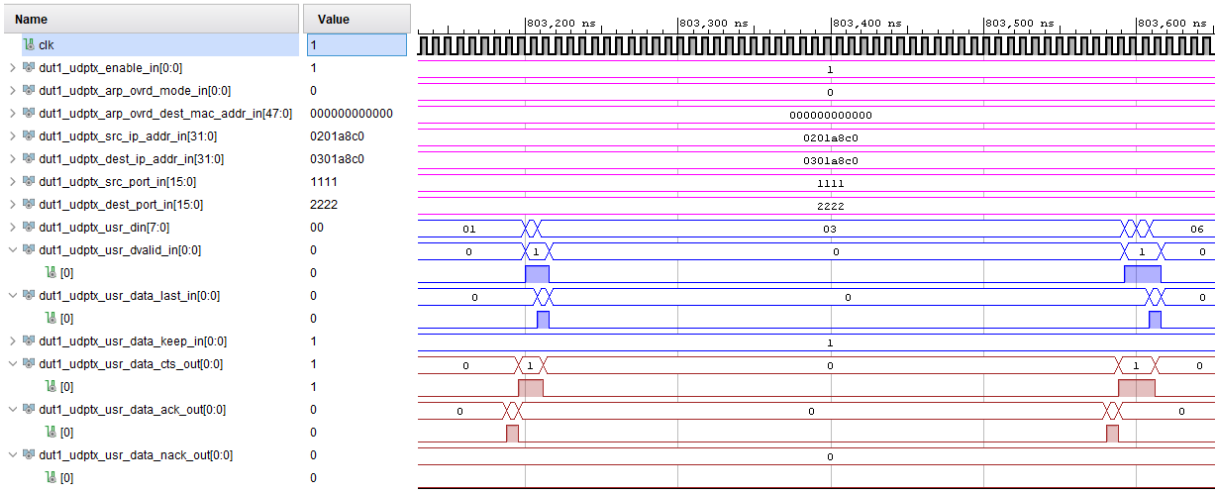

Figure 15- UDP TX Sending waveforms.

UDP RX Interface

The UDP RX interface sources UDP packets to the user as long as the packet's destination MAC and IP addresses match those of the IP core, regardless of the UDP source or destination ports.

Upon receiving an UDP packet, identification information including source and destination MAC address, the source and destination IP address, the source and destination UDP port is placed at corresponding output ports of the IP core. These values are valid from the beginning (first assertion of the tvalid signal) until the end (when both tvalid and tlast signals are asserted) of a received frame. They can be used by the user to filter the packets of interest. The UDP payload data are sourced to the user by the UDPRX Master AXIS interface. It is noted that the tready signal is missing in this interface, which means that user logic must be capable of handling the received frame without any back-pressure mechanism.

TCP Commands Interface

The TCP TX Slave AXIS interface (***tcptx_s_axis***) is dedicated for transmitting TCP segments. For receiving TCP segments, the TCP RX Master AXIS interface (***tcprx_m_axis***) is used. The users should rely on the basic handshake in an AXIS channel described above to handshake with these interfaces.

The TCP engine of the IP core can be configured either as a TCP server or a client. The IP core supports both active and passive TCP opening procedures as described in the RFC 793. In

order to open a TCP server to listen to a client connection request (in server mode) or to connect to a server (in client mode), the user should use the TCP command interface. The TCP opening procedure can be described as follows.

TCP Opening procedure

- **Step 1**: Configure initial parameters before opening a TCP session: ***tcptx_init_seq_num_in*** to indicate the initial sequence number (0 by default); ***tcptx_srv_mode_in*** to indicate whether we act as a server ('1') or client ('0'); ***tcptx_tcp_src_port_in*** to indicate our TCP port; ***tcptx_tcp_dest_port_in*** to indicate the peer's port (peer can be a server or a client, depending on our mode) and ***tcptx_server_ip_addr_in*** to indicate the server's IP address (in case we are in the client mode).
- **Step 2**: Wait for one clock period of the ***mac_tx_clk***.
- **Step 3**: wait until the signal ***tcptx_open_rdy_out*** to be asserted. The IP core is ready to open a TCP session.
- **Step 4**: Assert the signal ***async_tcptx_open_in*** to trigger the opening procedure.
- **Step 5**: Wait for at least 2 clock periods of the ***mac_tx_clk***.
- **Step 6**: De-assert the signal ***async_tcptx_open_in.***
- **Step 7**: Verify the connection establishment status reported by the signal ***tcptx_cntn_established_out.*** A timeout of several seconds can be implemented by the user. Upon timeout, if the connection has not been established, go to the closing procedure then restart all over again.

TCP Closing procedure

- **Step 1**: Assert the signal ***async_tcptx_abort_in.*** It is noted that this signal can be asynchronous to the IP clocks.
- **Step 2**: Wait for at least 2 clock periods of the ***mac_tx_clk.***
- **Step 3**: De-assert the signal ***async_tcptx_abort_in.***

Sometimes in the client mode the TCP opening procedure can fail, resulting on a timeout event described in **Step 7** of the Opening procedure. This is mostly because the TCP Server's MAC address is not known by the routing table. In such a case, the ARP-Cache component will send out an ARP request to discover the Server machine. Normally, if the server is online and responds to the ARP request, a second connection attempt should succeed.

IPCTEK

End Of Document.