TOE100G-IP Two Port reference design

Rev1.0 26-May-22

1 Introduction

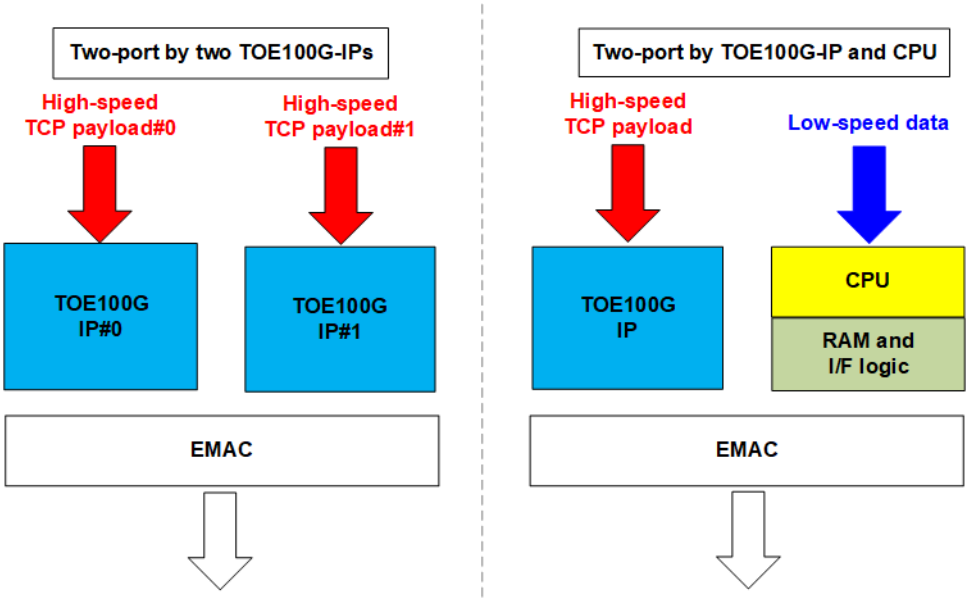

Figure 1-1 TCP/IP protocol layer

TOE100G-IP is designed to transfer TCP payload data at ultra-high speed by using one TCP port. However, there are some applications that need to use multiple ports for transferring many data types. Also, increasing the number of TCP ports can accelerate total transfer performance for transferring the data via one 100G Ethernet channel in some environments. If the additional ports are applied for transferring TCP payload data at high-speed rate or increase transfer performance for each 100G Ethernet channel, it is recommended to use multiple TOE100G-IPs in the system for handling each port independently, as shown in the left side of Figure 1-1.

However, some applications require only one fast port for transferring data at high-speed rate while the additional ports are applied for transferring some control information at low-speed rate with other protocols such as ICMP or DHCP. The right system of Figure 1-1 is purposed instead. One TOE100G-IP is applied for handling one high-speed TCP port while CPU is applied for handling the other ports or the other protocols with lower speed processing.

This document shows the right-side solution which uses CPU for handling other protocols - Ping command (ICMP protocol) and DHCP command (DHCP protocol). While one TOE100G-IP is integrated for processing one high speed TCP payload data.

2 Hardware overview

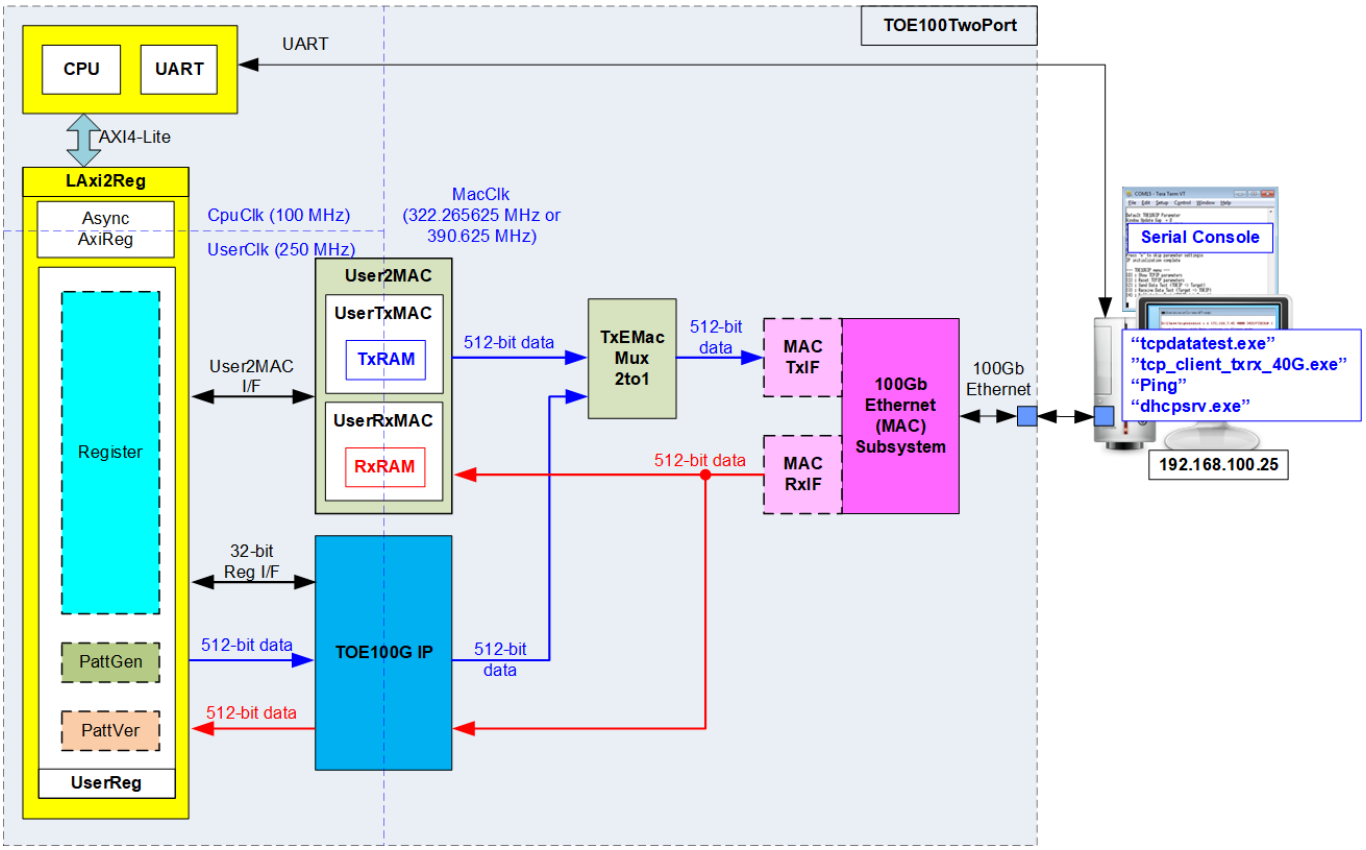

Figure 2-1 Demo block diagram

The reference design supports two port connections. First is slow-port connection which is handled by CPU system via User2MAC. The demo shows the slow-port connection for processing two protocols by using different CPU firmware – Ping test (ICMP protocol) and DHCP test (UDP protocol). For processing each protocol, CPU assigns different parameters to User2MAC via UserReg. Second is fast-port connection which is handled by TOE100G-IP. UserReg module connects to TOE100G-IP for setting the parameters via 32-bit Reg I/F and transferring high-speed data in both transfer directions by PattGen and PattVer. The transmitted packet from User2MAC and TOE100G-IP may be transferred in the same time. Therefore, TxEMacMux2to1 module is designed to be the switch logic to transfer the transmitted data from User2MAC and TOE100G-IP to Ethernet (MAC) subsystem. While the receive interface of Ethernet (MAC) subsystem is connected to both User2MAC and TOE100G-IP directly.

There are two 100Gb EMAC types – 100Gb Ethernet Subsystem and 100Gb Ethernet MAC Subsystem. For UltraScale+ device, 100Gb Ethernet Subsystem uses 512-bit AXI4 stream to be user data interface. Therefore, TOE100G-IP can connect with this module directly. While 100Gb Ethernet MAC subsystem on Versal device uses 384-bit AXI4 stream user interface. It needs to integrate MACTxIF and MACRxIF for converting data stream between 384 bits and 512 bits.

The target device in the test environment is the Test PC which can run four test applications. “tcpdatatest” and “tcp_client_txrx_40G” are the test application, provided by Design Gateway, for transferring high-speed data in half-duplex mode and full-duplex mode, respectively. “Ping” command is run to check round-trip time and “dhcpsrv” is run to assign IP address by DHCP (Dynamic Host Configuration Protocol)

There are three clock domains in the design, i.e., CpuClk which is the clock for running the CPU system, MacClk which is the user interface clock of 100Gb Ethernet (MAC) Subsystem, and UserClk which is the clock for running user logic of TOE100G-IP. In real system, the user can change the frequency of CpuClk and UserClk. According to TOE100G-IP datasheet, clock frequency of UserClk must be more than or equal to 220 MHz. AsyncAxiReg is designed to support asynchronous signals between CpuClk and UserClk. More details of each module inside the TOE100CPUTest are described as follows.

Note: When using 100G Ethernet Subsystem, MacClk is the output from 100G Ethernet subsystem and its frequency is equal to 322.266 MHz. While using 100G Ethernet MAC Subsystem, MacClk is generated by user logic and its frequency is equal to 390.625 MHz.

2.1 100G Ethernet (MAC) Subsystem (100G BASE-SR)

100G Ethernet (MAC) Subsystem implements the MAC layer and the low-layer protocol. To use 100G BASE-SR, physical connection of Ethernet cable is QSFP28 or 4xSFP28 connector. The IP core can be created by using IP wizard in Vivado tools.

For UltraScale+ device, 100G Ethernet Subsystem implements the MAC layer and PCS/PMA layer by integrating Transceiver module inside the IP. The user interface is 512-bit AXI4-stream at 322.266 MHz. More details of the core are described in the following link.

PG203: UltraScale+ Devices Integrated 100G Ethernet Subsystem Product Guide

https://www.xilinx.com/products/intellectual-property/cmac_usplus.html

For Versal device, 100G Ethernet MAC Subsystem implements the MAC layer and PCS logic without integrating Transceiver module. The user interface can be configured to several mode. In the reference design, Non-Segmented mode with independent clock is applied, so the user interface is 384-bit interface. The minimum clock frequency in this mode is 390.625 MHz. More details of the core are described in the following link.

PG314: Versal Devices Integrated 100G Multirate Ethernet MAC Subsystem Product Guide

https://www.xilinx.com/products/intellectual-property/mrmac.html

The user interface of 100G Ethernet MAC Subsystem in Versal device is different from the EMAC interface of TOE100G-IP. Therefore, the adapter logic for both Tx and Rx interface must be designed, as shown in Figure 2-2. More details of the adapter logic are described as follows.

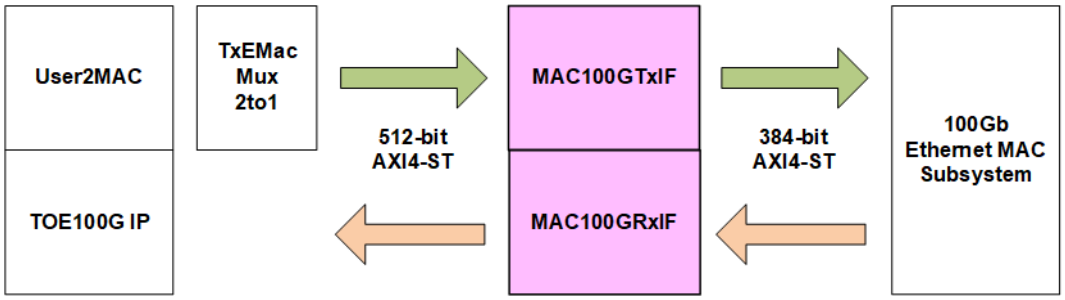

Figure 2-2 Adapter logic of EMAC interface in Versal Device

MAC100GTxIF

This module is AXI4-Stream converter from 512-bit to 384-bit to transfer data from user (TOE100G-IP) to 100G Ethernet MAC Subsystem. Therefore, it needs to have 384-bit register to store 128-bit user data that cannot be transmitted to EMAC for each cycle. rTempCnt is the control signal to show the amount of the unsent data in 128-bit unit which is stored in 384-bit internal register (rTempData). Four values are be assigned to show the amount of data, i.e., 000b (No data), 001b (has one 128-bit data), 011b (has two 128-bit data), and 111b (has three 128-bit data or full). The format of data output to EMAC is mixed signal of user data (U2MACData) and 384-bit rTempData, controlled by rTempCnt. Timing diagram to show more details of MAC100GTxIF is are shown in Figure 2-3.

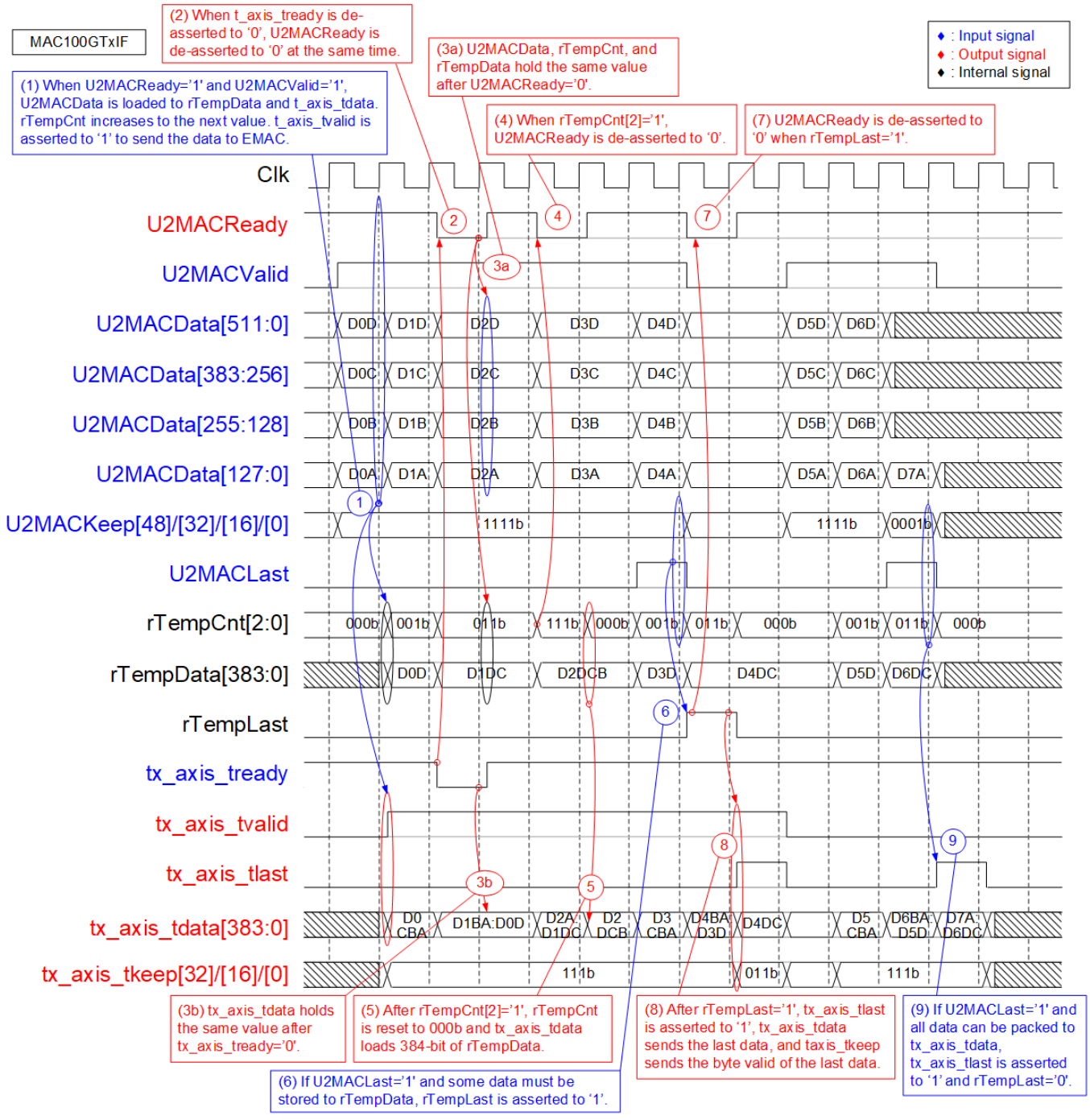

Figure 2-3 MAC100GTxIF Timing diagram

1. When the first data is received from user (U2MACValid='1' and U2MACReady='1' while rTempCnt=000b), 384-bit user data (U2MACData) is transferred to EMAC. tx_axis_tvalid is asserted to '1' and tx_axis_tdata loads 384-bit data from U2MACData. If this data is not the last data, the upper 128-bit unsent data is stored to internal register (rTempData) and rTempCnt increases the value by the sequence: 000b -> 001b -> 011b -> 111b. Also, tx_axis_tkeep are asserted to all one to transfer 384-bit data to EMAC.
2. When EMAC is not ready, tx_axis_tready is de-asserted to '0'. U2MACReady is de-asserted to '0' to pause user data transmission.
3. When tx_axis_tready and U2MACReady are de-asserted to '0', the output signals to EMAC (tx_axis_tvalid, tx_axis_tlast, tx_axis_tdata, and tx_axis_tkeep) and the input signals from user (U2MACValid, U2MACData, U2MACKeep, and U2MACLast) must hold the same value until the ready signals are re-asserted to '1' to accept the current data.
4. When 384-bit register (rTempData) stores three 128-bit data, rTempCnt is equal to 111b (full condition). At this time, U2MACReady is de-asserted to '0' to pause user data transmission. After that, 384-bit data of rTempData is flushed to EMAC.
5. "tx_axis_tdata" loads 384-bit data from rTempData and then rTempCnt is reset to 000b. There is no unsent data stored in rTempData.
6. The last user data is transmitted with asserting U2MACLast to '1'. U2MACKeep is read to check the number of valid bytes of the last data. Also, rTempCnt is read to check the amount of unsent data. In the example, one 128-bit data is stored in rTempCnt and 512-bit user data is received, so two 128-bit data must be stored to rTempCnt. In this case, rTempLast is asserted to '1' to store the unsent last data.
   Note: Step 9) shows the example when the last user data is received but all data can be transferred to EMAC without storing any data in rTempData.
7. rTempLast is asserted to '1' when the last user data is stored in rTempData. At the same time, U2MACReady is de-asserted to '0' to pause user data transmission.
8. The last data from rTempData is transferred to EMAC. tx_axis_tlast is asserted to '1' and tx_axis_tkeep shows the amount of valid byte.
9. This step shows the example when there are two 128-bit data stored in rTempData and 128-bit last data is transmitted by user. Therefore, total data which has three 128-bit data can be transferred to tx_axis_tdata with asserting tx_axis_tlast to '1'. There is no data stored in rTempData and rTempLast is not asserted to '1'.

MAC100GRxIF

This module is AXI4-Stream converter from 384-bit to 512-bit to transfer data from 100G Ethernet MAC Subsystem to user (TOE100G-IP). The logic includes Latch register to store the unsent data that is not transferred to user. To support data realignment, three counters are designed to check the amount of data in 128-bit unit. First counter is wRx128bDataCnt which shows the amount of received data from EMAC which can be equal to 1, 2, or 3. Second counter is rLatDataCnt which shows the amount of unsent data that is received from EMAC. The unsent data is stored to the latch register (rDataLat). This counter can be equal to 0 – 3. Last counter is wRxTotalDataCnt which shows the sum of the amount of received data and the unsent data (wRx128bDataCnt + rLatDataCnt). Therefore, the value of the third counter can be equal to 1 – 6. When wRxTotalDataCnt is more than or equal to 4 (5 or 6), 256-bit data will be packed and transmitted to user. However, the last data transmitted to the user can be less than 256-bit data by controlling byte enable value (MAC2UKeep).

The second counter (rLatDataCnt) is updated by several conditions.

(1) When the first data is received and there is no unsent data stored in rDataLat (rLatDataCnt=0), all bits of the first data is loaded to rDataLat. Therefore, rLatDataCnt must be equal to the amount of received data from EMAC (wRx128bDataCnt or wRxTotalDataCnt which is the same value when rLatDataCnt=0).
(2) When total amount of data (wRxTotalDataCnt) is more than or equal to 4, one 512-bit data will be transmitted to TOE100G-IP. Therefore, the amount of unsent data (rLatDataCnt) is decreased by 4 (wRxTotalDataCnt – 4).
(3) When the last data is transmitted and there is no new packet is received, Latch register now is empty status. Therefore, rLatDataCnt is reset to 0.
(4) There is a special case that the last data is transmitted while the first data of the new packet is received. This condition is combination of (1) and (3). Therefore, the amount of unsent data is equal to the amount of received data in the new packet (wRx128bDataCnt).

384-bit latch register (rDataLat) uses rLatDataCnt to determine the maximum number of received data from EMAC that must be kept in the next cycle.

(1) When rLatDataCnt = 0, three 128-bit new data (rx_axis_tdata[384:0]) must be kept.
(2) When rLatDataCnt = 3, one 128-bit new data can be packed with three 128-bit previous data (rDataLat[383:0]). Therefore, two 128-bit new data (rx_axis_tdata[384:128]) must be kept to rDataLat.
(3) When rLatDataCnt = 2, two 128-bit new data can be packed with two 128-bit previous data (rDataLat[255:0]). Therefore, one 128-bit data (rx_axis_tdata[384:256]) must be kept to rDataLat.
(4) When rLatDataCnt = 1, all new data can be packed with one 128-bit previous data (rDataLat[127:0]). There is no data kept to rDataLat.

To transfer the last data from EMAC to the user, it has two behaviors.

(1) If all last data from EMAC can be packed with rDataLat (wRxTotalDataCnt ≤ 4), the last data will be transferred to the user in the next cycle.
(2) When wRxTotalDataCnt of the last data is more than 4, it needs two cycles to send all data – 512-bit data for the 1st cycle and the remained data for the 2nd cycle. To support this feature, rExLast is designed to latch the last flag of EMAC for sending the last data in the 2nd cycle.

Timing diagram to show MAC100GRxIF operation is shown in Figure 2-4.

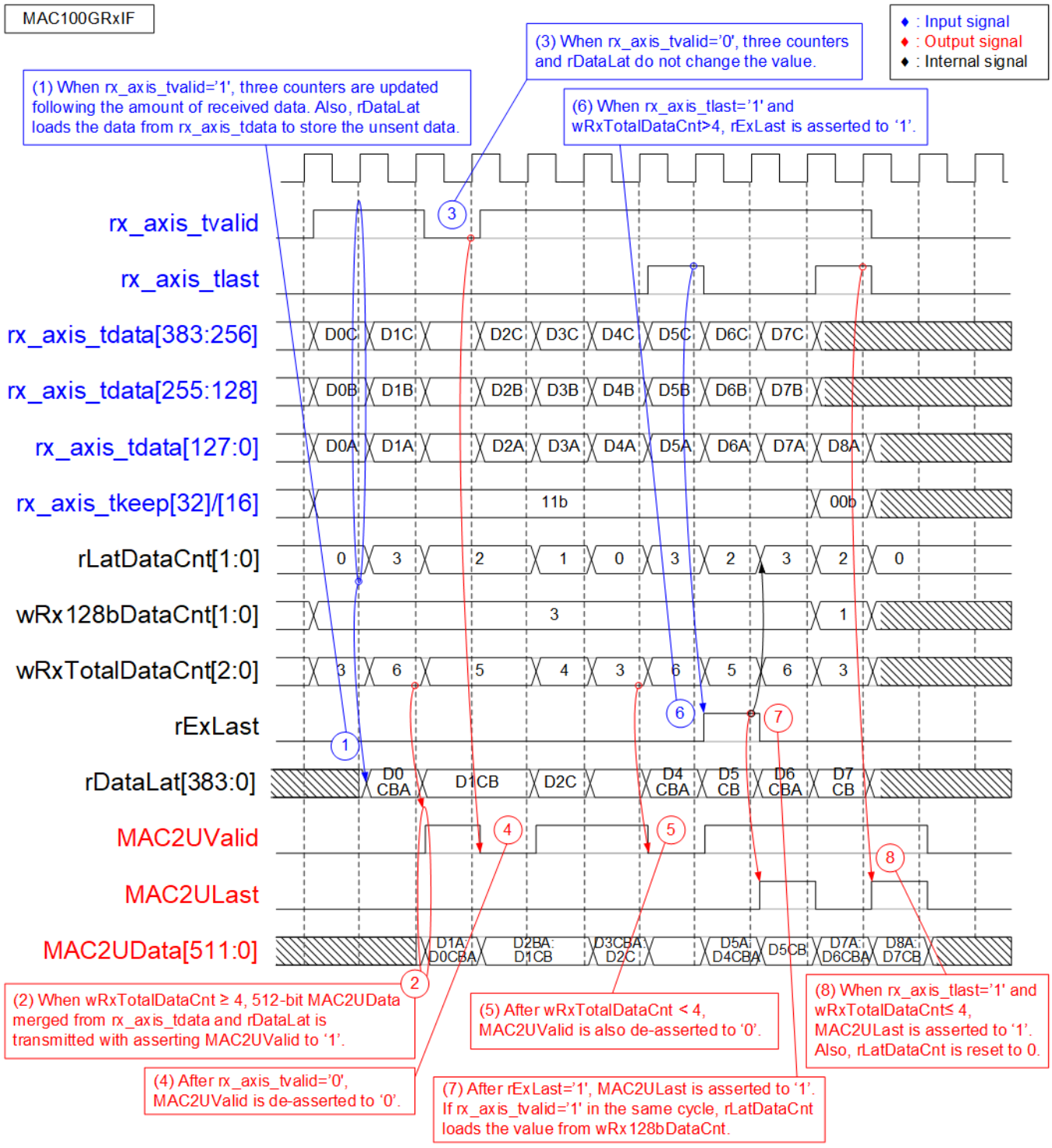

Figure 2-4 MAC100GRxIF Timing diagram

1. When the new 384-bit data is received from EMAC, wRx128bDataCnt is equal to 3. If the previous clock cycle is Idle, rLatDataCnt is equal to 0. Therefore, wRxTotalDataCnt is equal to 3 (0+3). When rLatDataCnt=0, all 384-bit data is loaded to rDataLat. As wRxTotalDataCnt is less than 4, there is no data transmitted to MAC2U I/F.
2. Next, 384-bit data are received from EMAC while rLatDataCnt is equal to 3 (the amount of data that is stored to rDataLat in the previous clock cycle). Therefore, wRxTotalDataCnt is equal to 6 (3 + 3) which is enough to transmit the data to MAC2U I/F. MAC2UValid is asserted to '1' to send 512-bit M2UData and M2UData loads three 128-bit data (D0A, D0B, and D0C) from rDataLat and one 128-bit data from rx_axis_tdata (D1A). Therefore, two 128-bit data (D1B and D1C) are unsent and stored to rDataLat. rLatDataCnt is updated to 2.
3. EMAC de-asserts rx_axis_tvalid to '0' when it is not ready to transmit the new data. Three counters (rLatDataCnt, wRx128bDataCnt, and wRxTotalDataCnt) and rDataLat hold the same value to wait more data from EMAC.
4. If EMAC pauses data transmission by de-asserting rx_axis_tvalid to '0', MAC2UValid is de-asserted to '0' in the next clock.
5. At the first cycle of every four cycle to receive 384-bit data from EMAC, rLatDataCnt is equal to 0 and wRxTotalDataCnt is less than 4. Therefore, MAC2UValid is de-asserted to '0' to pause data transmission to the user.
6. When the last data is received from EMAC (rx_axis_tlast='1' and rx_axis_tvalid='1') and wRxTotalDataCnt in that cycle is more than 4 (5 or 6), the latch flag to store last signal (rExLast) is asserted to '1'. At the same time, 512-bit data is transferred to the user while the remained last data is transferred in the next cycle.
7. After rExLast is asserted to '1', MAC2ULast is asserted to '1' to send the remained last data which is stored in rDataLat. If the first data of the new packet is transferred from EMAC immediately, the first data is loaded to rDataLat and rLatDataCnt is equal to the amount of 128-bit data in the first cycle.
8. The example to send the last data without asserting rExLast is shown in this step. When rx_axis_tlast is asserted to '1' and wRxTotalDataCnt is less than or equal to 4, the last data can be packed and transferred to the user in the next cycle. Therefore, rExLast is not asserted to '1' and rLatDataCnt is reset to 0.

2.2 TxEMacMux2to1

This module is data switch to select transmitted data from two sources to forward to EMAC. rChSel is the control signal to select the active channel. When two channels are requested in Idle condition, the same channel is selected. After finishing the current channel transferring, rChSel changes the value to transfer the data from another channel. More details are shown in Figure 2-5.

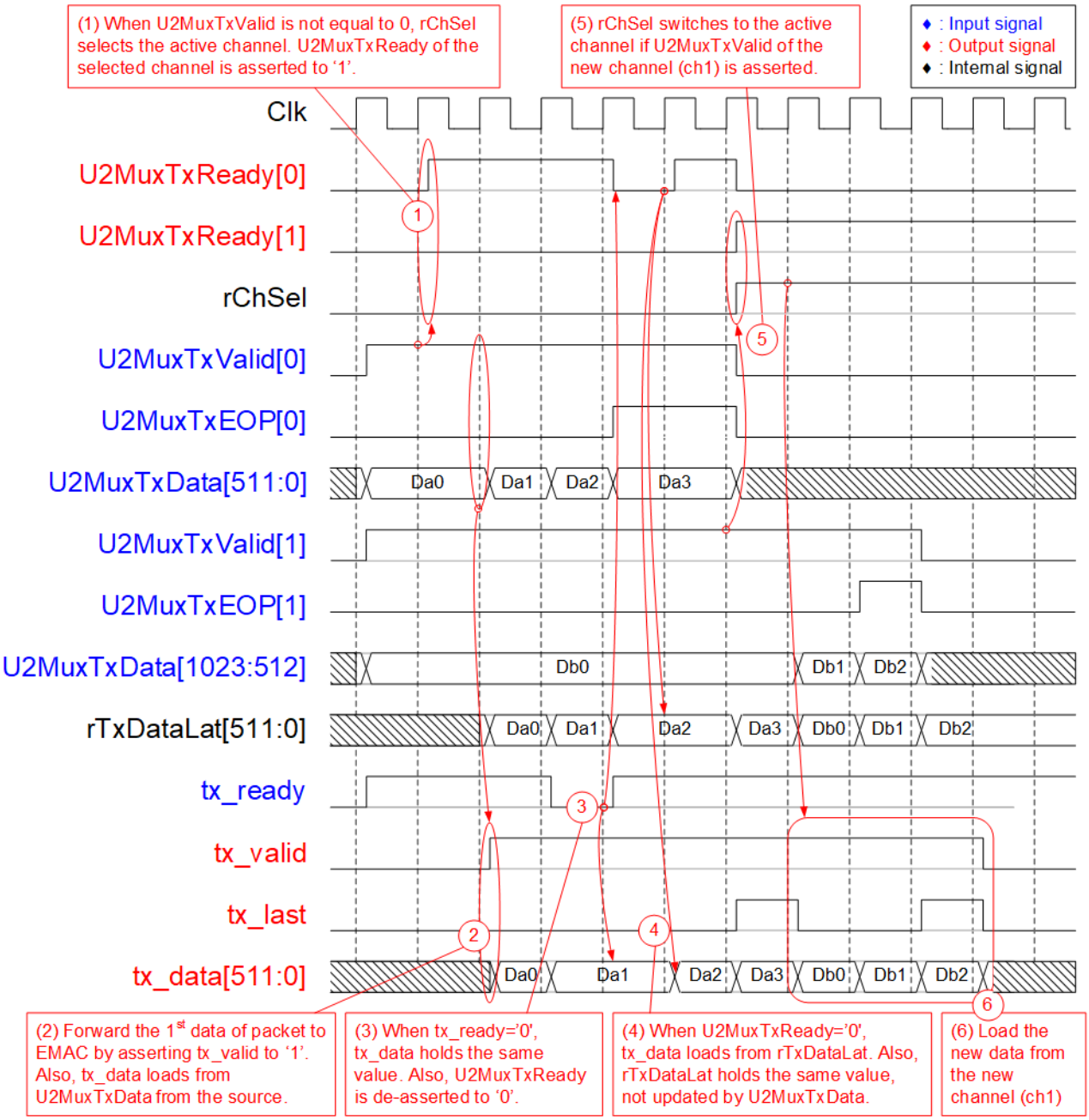

Figure 2-5 TxEMacMux2to1 timing diagram

1. When two user sends the new packet by asserting U2MuxTxValid to '1' at the same time and now the module is in Idle condition, rChSel (the signal to indicate the active channel) does not change the value to forward the data from the same channel to EMAC. In Figure 2-5, the Ch#0 is selected, so U2MuxTxReady of the selected channel (Ch#0) is asserted to '1' to accept the first data.
2. The input signals of the selected channel (Ch#0), i.e., U2MuxTxEOP#0 (end-of-packet) and U2MuxTxData[511:0] (512-bit data) are loaded to the output signals of EMAC (tx_last and tx_data, respectively). Also, tx_valid is asserted to '1' to start sending the new packet to EMAC.
3. When EMAC is not ready to receive data by de-asserting tx_ready to '0', all output signals of EMAC hold the same value. Also, U2MuxTxReady of the active channel is de-asserted to '0' to hold the input signals from the source.
4. After EMAC re-asserts tx_ready to accept the data, the next output signals to EMAC will be loaded from the internal latch register (rTxDataLat). The internal latch register is designed to load the data from the active source when U2MuxTxReady is asserted to '1'. Therefore, the latch register is applied to store the unsent data to EMAC when EMAC pauses data transmission.
5. After accepting the final data of a packet from the active channel, the next active channel is scanned. If U2MuxTxValid of another channel is asserted, rChSel will switch the value. In Figure 2-5, the next active channel is Ch#1 (rChSel='1') to accept the data from Ch#1.
6. The input signals (U2MuxTxEOP and U2MuxTxData) of the active channel (Ch#1) are forwarded to be the output signals of EMAC (tx_last and tx_data) until transferring the final data of a packet

2.3 TOE100G-IP

TOE100G-IP implements TCP/IP stack and offload engine. User interface has two signal groups, i.e., control signals and data signals. Register interface is applied to set control registers and monitor status signals. Data signals are accessed by using FIFO interface. The interface with 100G EMAC is 512-bit AXI4 interface.

More details are described in datasheet.
https://dgway.com/products/IP/TOE100G-IP/dg_toe100gip_data_sheet_xilinx.pdf

2.4 User2MAC

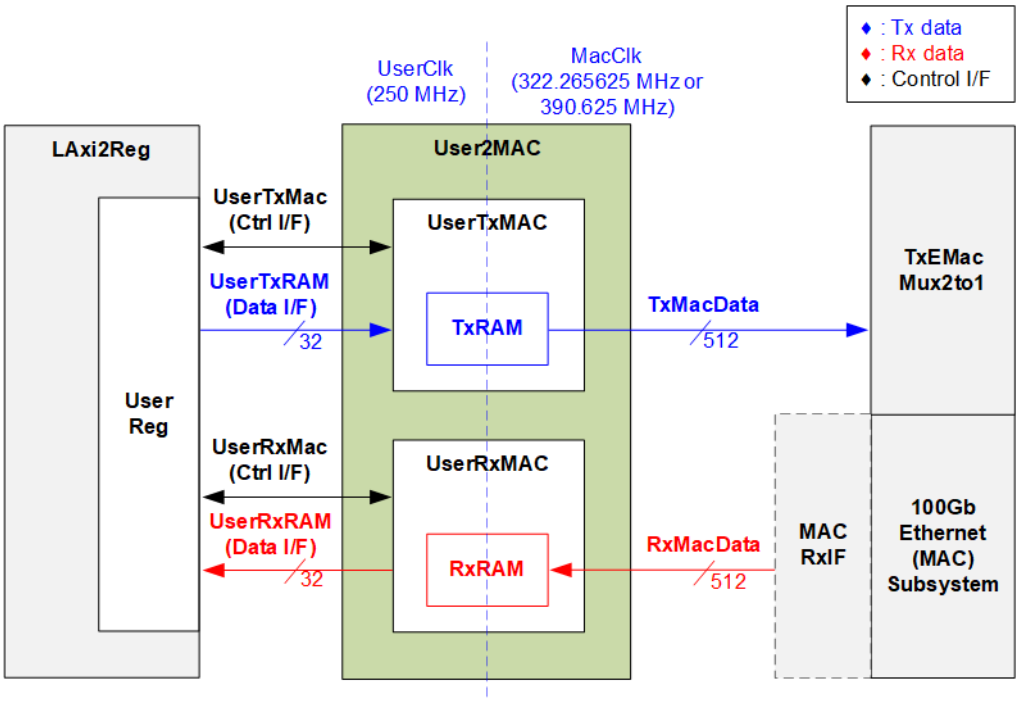

Figure 2-6 User2MAC block diagram

User2MAC is responsible to transfer Ethernet packet by using low-speed connection. Two protocols are implemented in the reference design. First is Ping command which is ICMP protocol for checking round-trip time. ICMP echo reply packet is returned after receiving ICMP echo request packet. Second is DHCP for dynamic IP address assignment. To operate DHCP, UDP protocol is configured to User2MAC. The Ethernet packet is created and decoded by CPU via LAxi2Reg. Data bus width on LAxi2Reg side is 32 bits while data bus width on MAC I/F is 512 bits.

User2MAC consists of two modules, i.e., UserTxMAC and UserRxMAC. UserTxMAC includes TxRAM for storing Ethernet packet that is prepared by CPU firmware. While UserRxMAC includes RxRAM for storing Ethernet packet from EMAC. Before storing the packet to RxRAM, there is filtering logic to verify some Ethernet data. Only the valid Ethernet packet is stored to RxRAM while the invalid packet is rejected. After that, CPU reads RxRAM to decode the packet. More details of UserTxMAC and UserRxMAC are described as follows.

UserTxMAC

UserTxMAC includes 64 x 512-bit Asynchronous dual port RAM to store the transmitted packet that is written by CPU via UserTxRam write I/F. Packet size (UserTxMacLen) is also set by CPU. After CPU asserts the request (UserTxMacReq), the logic starts forwarding the packet read from TxRAM to EMAC. Transmit interface of EMAC is 512-bit AXI4 stream which may de-assert ready (TxReady) to pause data transmission. After finishing the packet transmission to EMAC, busy signal (UserTxMacBusy) is de-asserted to '0'. More details about the logic design inside UserTxMAC is shown in Figure 2-7.

Note: UserTxRam Write I/F with CPU uses 32-bit data while TxRAM data width is 512-bit data. Therefore, there is decoder to create write byte enable to write only some bytes of 512-bit data bus of TxRAM.

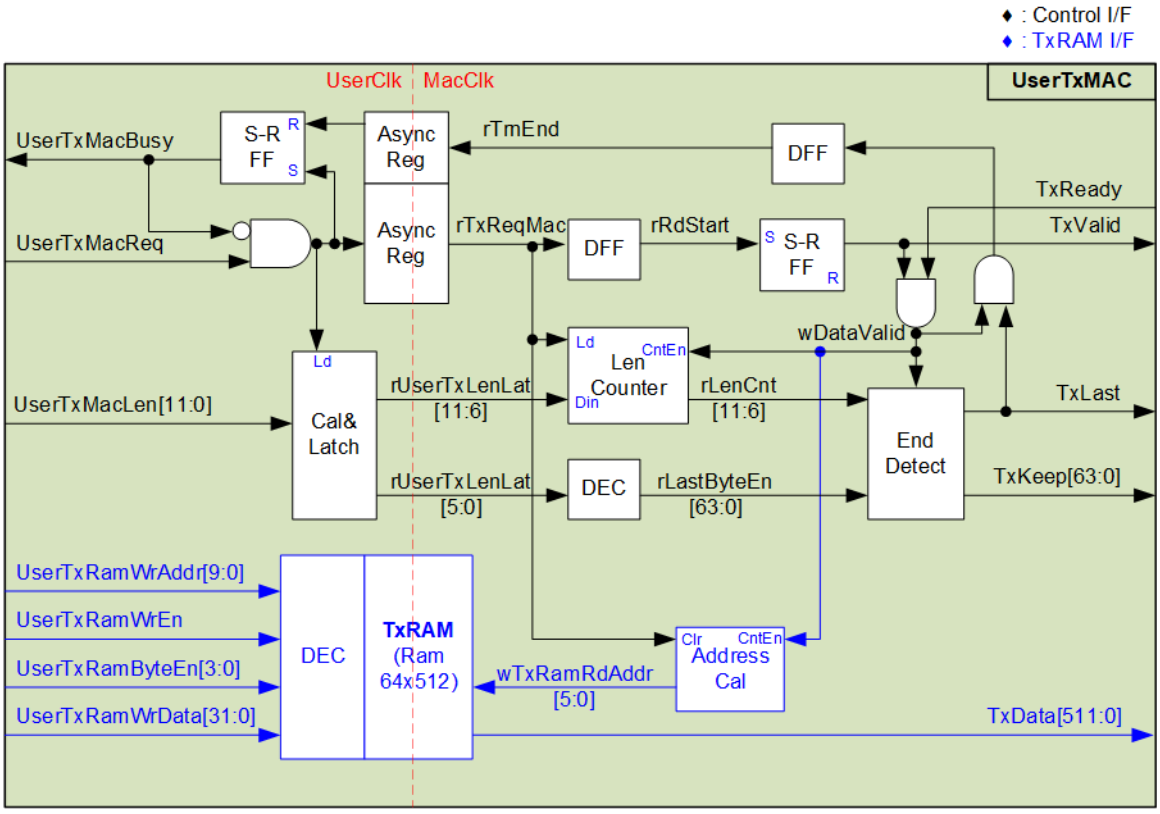

Figure 2-7 UserTxMAC Logic Diagram

The steps to transmit a packet from UserTxMAC are described as follows.

1. CPU checks that UserTxMacBusy='0' to confirm UserTxMAC is in Idle state.
2. CPU prepares a transmitted packet to TxRAM. The first data is written at address#0. (UserTxRamWrAddr=0). Therefore, the maximum transmitted packet size is 4 Kbytes which is the size of TxRAM.
   Note: TxRAM has byte enable to allow CPU to write TxRAM by using byte unit.
3. CPU sets UserTxMacLen to set transmit packet size in byte unit. Also, UserTxMacReq is asserted to '1' to begin data transmission.
4. After that, the request signal is transferred to MacClk domain via AsyncReg module. Also, UserTxMacBusy is asserted to '1' to show user that the operation is run. Total transfer size (UserTxMacLen) is loaded to internal logic. The transfer size is split into two parts. First is the amount of 512-bit data which is calculated by using UserTxMacLen[11:6]. The value is rounded up when the size is not aligned to 512-bit. Another part is bit[5:0] which is latched to create the byte enable of the final data of a packet (rLastByteEn and TxKeep) by using decoder.
5. When the start flag on MacClk domain is asserted, the first data is read from TxRAM and the data valid (TxValid) is asserted to '1'. The read address (wTxRamRdAddr) is up-counted to transfer the next data from TxRAM after finishing each data transferring (TxValid='1' and TxReady='1'). Also, the Length counter (rLenCnt) is down-counted when finishing transferring each data to check the last data position.
6. When rLenCnt=1 or the next data is the final data of a packet, last flag (TxLast) is asserted to '1'. Also, byte enable (TxKeep) loads the last value from rLastByteEn. TxKeep is equal to all one to send 512-bit data when the data is not the final data of a packet.
7. After the final data is transferred, End flag (rTrnEnd) is asserted for de-asserting busy flag. It needs to pass AsyncReg to transfer the End flag from MacClk to Clk domain.

UserRxMAC

UserRxMAC has three operations to validate the received packet and store the valid packet to RxRAM which is 64 x 512-bit Asynchronous dual port RAM. Therefore, the logic inside UserRxMAC can be divided to three groups. First is the logic to verify 38-byte header (byte#0 – byte#37) of each received packet. The expected value and mask bit to enable data comparison are set by user. Only the packet that includes the correct header can be accepted. Second is the logic to check enable flag from user and free space in RxMacFf. The packet will be rejected if the user disable this module or FIFO has not enough space. RxMacFf is applied to store the end address of RxRAM after finishing storing the received packet. Therefore, CPU determines the received packet size from the end address. Third is the logic to store the received packet to RxRAM. More details about the logic design inside UserRxMAC is shown in Figure 2-8.

Note: UserRxRam Read I/F with CPU uses 32-bit data while RxRAM data width is 512-bit data. Therefore, 16-to-1 Mux is integrated to select 32-bit data from 512-bit data. Read latency.

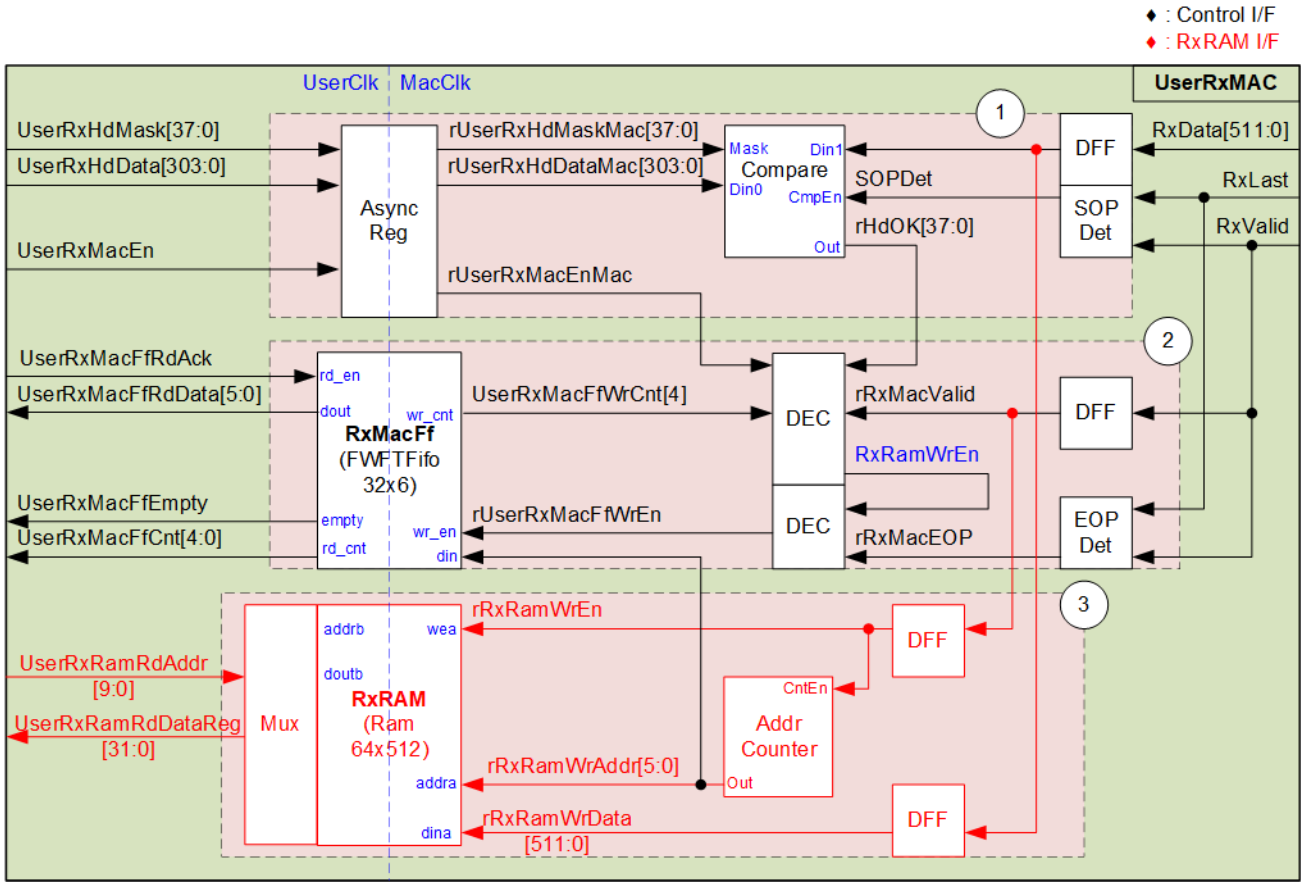

Figure 2-8 UserRxMAC Logic Diagram

As shown in Figure 2-8, Block (1) is the logic to verify 38-byte header of each packet. Block (2) is the logic to store the end address of RxRAM and Block (3) is the logic to store the received packet. The details of UserRxMAC operation when the packet is received are described as follows.

1. Two parameters are configured by user – 38-byte header data (UserRxHdData) and 38-bit data mask to verify the packet header (UserRxHdMask). Both parameters must be stable when user enables this module by asserting UserRxMacEn to '1'. The parameters must be forwarded to AsyncReg for clock domain crossing. When the first data of the new packet is received, SOPDet asserts the signal to start Compare module. 38-byte header data is compared to byte#0 - byte#37 of received data, controlled by data mask bit. One bit of data mask is referred to one byte of received data. If data mask is de-asserted to '0', that data byte is bypassed. Therefore, header verification is disabled if UserRxHdMask is set to all zero. When the header of received packet is valid, rHdOK is asserted to '1'. The packet can be stored to RxRAM only when 38-bit of rHdOK is equal to all one.
2. Next, two signals are additional read - enable flag from user (UserRxMacEn) and RxMacFf data counter (UserRxMacFfWrCnt). It needs to confirm that CPU is ready for processing the received packet by asserting UserRxMacEn to '1' and the logic has enough free space to store received packet and write pointer of RxRAM. Bit4 of UserRxMacFfWrCnt must be equal to 0. If both conditions are met and the header is valid, write enable of RxRAM (RxRamWrEn) is asserted to store the received data to RxRAM. RxMacFf stores the RAM address after finishing storing each packet to RxRAM. Therefore, EOPDet is designed to assert a pulse when end of packet is received. After that, rUserRxMacFfWrEn is asserted to '1' to write end address to RxMacFf.
3. Last, the valid packet is stored to RxRAM by asserting write enable to RxRAM (rRxRamWrEn) when the data is received (RxValid='1'). The address counter is counted after each 512-bit data is stored to RxRAM. The last address after receiving end-of-packet is stored to RxMacFf.
   Note: Using Bit4 of UserRxMacFfWrCnt for checking FIFO space, up to 16 addresses can be stored to RxMacFf. Therefore, up to 16 packets can be stored to RxRAM. Since RxRAM size is 4 Kbyte, one packet size should be not more than 256 bytes. However, the user can modify RAM size and FIFO size to match with user system requirement.

The CPU function for processing the received packet that stores in UserRxMAC is described as follows.

1. CPU waits until FIFO is not empty (UserRxMacFfEmpty='0').
2. CPU read the last address via UserRxMacFfRdData signal which is valid for read because RxMacFf is FWFT FIFO.
3. After that, CPU asserts UserRxMacFfRdAck to '1' to flush the read data from RxMacFf.
4. CPU reads and decodes one received packet from RxRAM, starting from the latest read position to the last address that reads from RxMacFf. After finishing packet processing, CPU returns to step 1 to wait and process the next packet.
   Note: UserRxRamRdAddr is the address for 32-bit data while rRxRamWrAddr is the address for 512-bit data. Therefore, CPU firmware must convert the 512-bit address that is stored in RxMacFf to 32-bit address before starting reading data from RxRAM.

2.5 CPU and Peripherals

32-bit AXI4-Lite is applied to be the bus interface for the CPU accessing the peripherals such as Timer and UART. To control and monitor the test system, the control and status signals are connected to register for CPU access as a peripheral through 32-bit AXI4-Lite bus. CPU assigns the different base address and the address range to each peripheral for accessing one peripheral at a time.

In the reference design, the CPU system is built with one additional peripheral to access the test logic. So, the hardware logic must be designed to support AXI4-Lite bus standard for supporting CPU writing and reading. LAxi2Reg module is designed to connect the CPU system as shown in Figure 2-9.

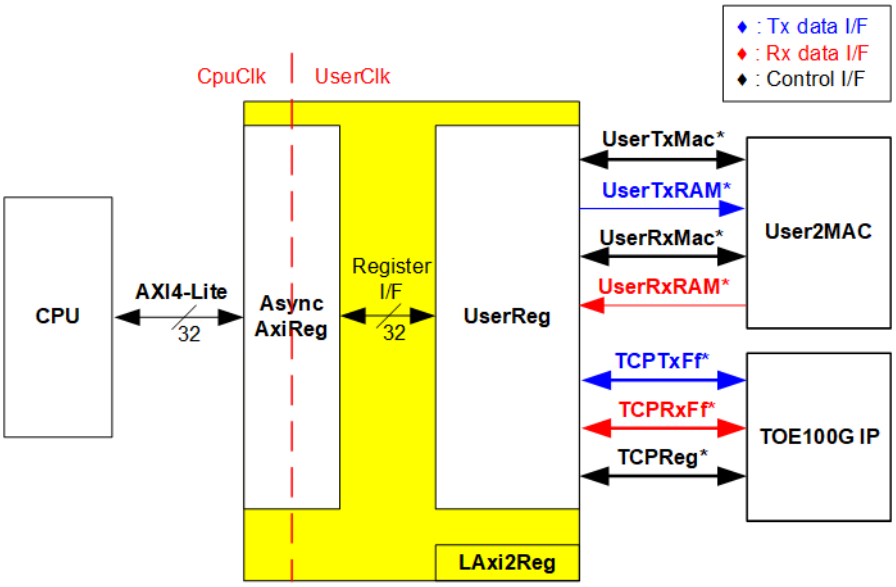

Figure 2-9 LAxi2Reg block diagram

LAxi2Reg consists of AsyncAxiReg and UserReg. AsyncAxiReg is designed to convert the AXI4-Lite signals to be the simple register interface which has 32-bit data bus size (similar to AXI4-Lite data bus size). Also, AsyncAxiReg includes asynchronous logic to support clock domain crossing between CpuClk and UserClk.

UserReg has the register files for storing the parameters which are set to User2MAC and TOE100G-IP. Also, the status signals of User2MAC and TOE100G-IP are mapped for CPU reading. The data interface of User2MAC uses simple dual-port RAM interface while the data interface of TOE100G-IP uses FIFO interface. More details of AsyncAxiReg and UserReg are described as follows.

2.5.1 AsyncAxiReg

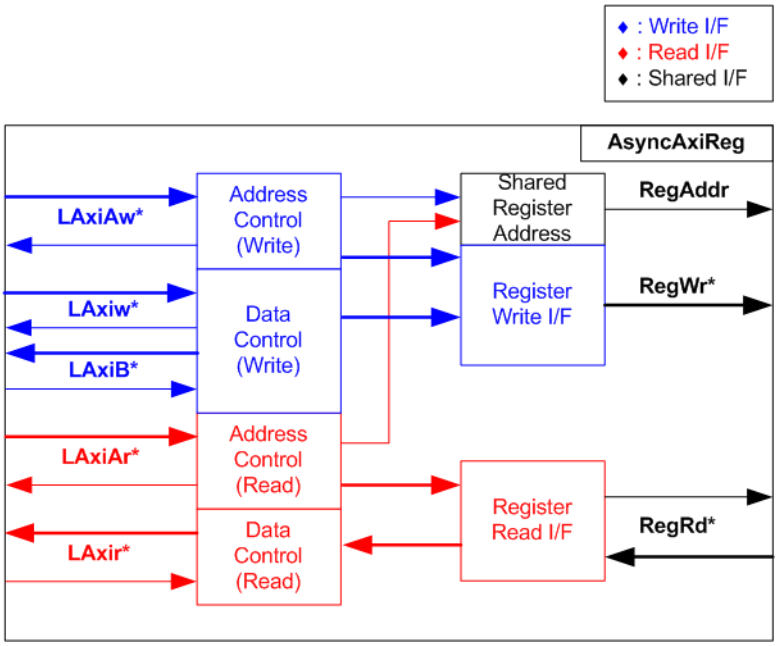

Figure 2-10 AsyncAxiReg interface

The signal on AXI4-Lite bus interface can be split into five groups, i.e., LAxiAw* (Write address channel), LAxiw* (Write data channel), LAxiB* (Write response channel), LAxiAr* (Read address channel), and LAxir* (Read data channel). More details to build custom logic for AXI4-Lite bus is described in following document.
https://forums.xilinx.com/xlnx/attachments/xlnx/NewUser/34911/1/designing_a_custom_axi_slave_rev1.pdf

According to AXI4-Lite standard, the write channel and the read channel are operated independently. Also, the control and data interface of each channel are run separately. The logic inside AsyncAxiReg to interface with AXI4-Lite bus is split into four groups, i.e., Write control logic, Write data logic, Read control logic, and Read data logic as shown in the left side of Figure 2-10. Write control I/F and Write data I/F of AXI4-Lite bus are latched and transferred to be Write register interface with clock domain crossing registers. Similarly, Read control I/F of AXI4-Lite bus are latched and transferred to be Read register interface. While the returned data from Register Read I/F is transferred to AXI4-Lite bus by using clock domain crossing registers. In register interface, RegAddr is shared signal for write and read access, so it loads the address from LAxiAw for write access or LAxiAr for read access.

The simple register interface is compatible with single-port RAM interface for write transaction. The read transaction of the register interface is slightly modified from RAM interface by adding RdReq and RdValid signals for controlling read latency time. The address of register interface is shared for write and read transaction, so user cannot write and read the register at the same time. The timing diagram of the register interface is shown in Figure 2-11.

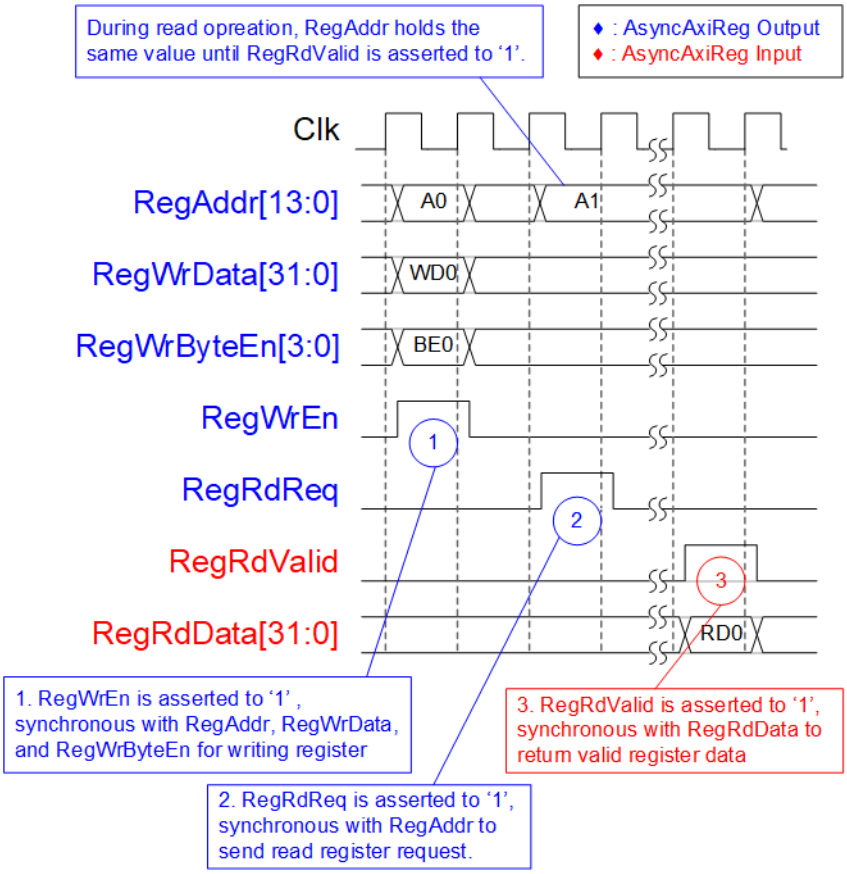

Figure 2-11 Register interface timing diagram

1) To write register, the timing diagram is similar to single-port RAM interface. RegWrEn is asserted to '1' with the valid signal of RegAddr (Register address in 32-bit unit), RegWrData (write data of the register), and RegWrByteEn (the write byte enable). Byte enable has four bits to be the byte data valid. Bit[0], [1], [2], and [3] are equal to '1' when RegWrData[7:0], [15:8], [23:16], and [31:24] are valid, respectively.
2) To read register, AsyncAxiReg asserts RegRdReq to '1' with the valid value of RegAddr. 32-bit data must be returned after receiving the read request. The slave must monitor RegRdReq signal to start the read transaction. During read operation, the address value (RegAddr) does not change the value until RegRdValid is asserted to '1'. So, the address can be used for selecting the returned data by using multiple layers of multiplexer.
3) The read data is returned on RegRdData bus by the slave with asserting RegRdValid to '1'. After that, AsyncAxiReg forwards the read value to LAxir* interface.

2.5.2 UserReg

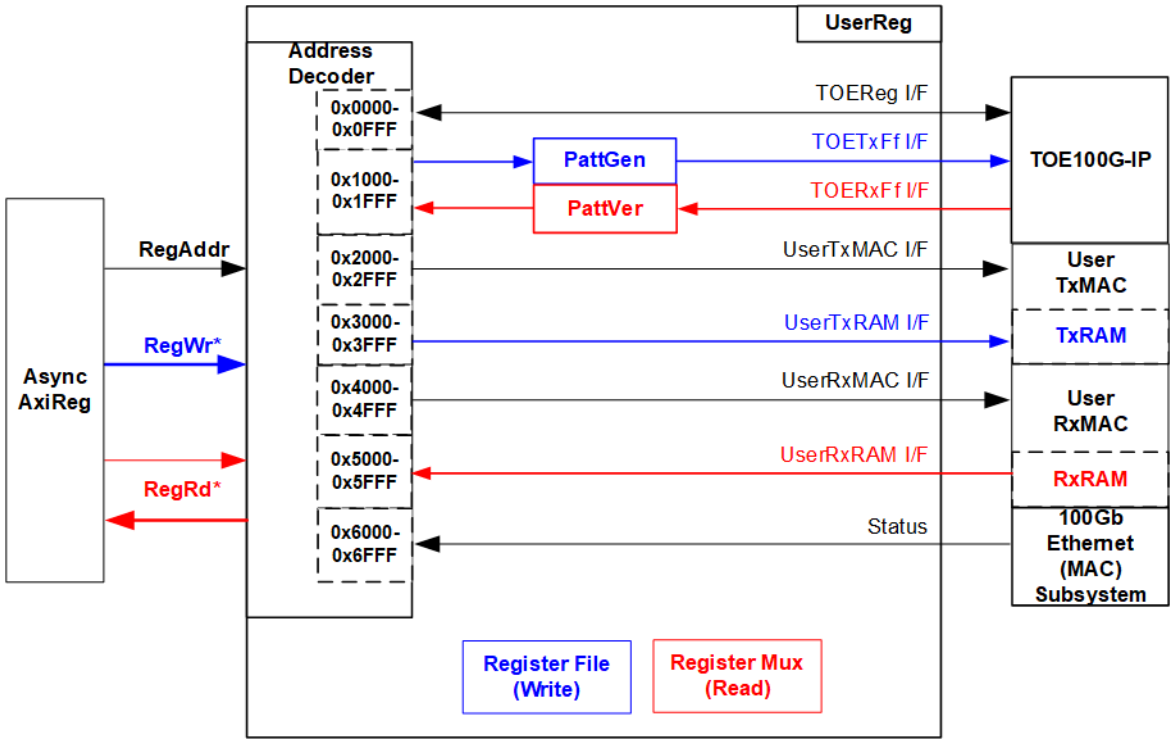

Figure 2-12 UserReg block diagram

The logic inside UserReg has three operations, i.e., Address decoder with Register File for write access and Register Mux for read access, Pattern generator (PattGen), and Pattern verification (PattVer). More details are described as follows.

Address decoder with Register File and Register Mux

As shown in Figure 2-12, the address range, mapped to UserReg, is split into seven areas.

1) 0x0000 – 0x0FFF: Register interface of TOE100G-IP
2) 0x0000 – 0x1FFF: PattGen and PattVer signals for transferring data with TOE100G-IP
3) 0x2000 – 0x2FFF: Control and status signals of UserTxMAC
4) 0x3000 – 0x3FFF: Write interface of Tx RAM inside UserTxMAC
5) 0x4000 – 0x4FFF: Control and status signals of UserRxMAC
6) 0x5000 – 0x5FFF: Read interface of Rx RAM inside UserRxMAC
7) 0x6000 – 0x6FFF: Status signal of EMAC

The upper bits of RegAddr are applied to select the active module for writing or reading while the lower bits of RegAddr are forwarded to each module to access the internal signals within each module. The details of register map are shown in Table 2-1.

To read register, it includes many multiplexers to select the data from each module. Therefore, the read latency time is increased from the multiplexer. The slowest path to return read data is the read data from UserRxRAM which has four clock cycles latency time. Therefore, RegRdValid is created by RegRdReq with asserting four D Flip-flops.

Table 2-1 Register map Definition

Address Wr/Rd	Register Name (Label in the "ping100gtest.c" and "dhcp100gtest.c")	Description
BA+0x0000 – BA+0x00FF: TOE100G-IP Register Area More details of each register are described in TOE100G-IP datasheet.		
BA+0x0000	TOE_RST_INTREG	Mapped to RST register within TOE100G-IP
BA+0x0004	TOE_CMD_INTREG	Mapped to CMD register within TOE100G-IP
BA+0x0008	TOE_SML_INTREG	Mapped to SML register within TOE100G-IP
BA+0x000C	TOE_SMH_INTREG	Mapped to SMH register within TOE100G-IP
BA+0x0010	TOE_DIP_INTREG	Mapped to DIP register within TOE100G-IP
BA+0x0014	TOE_SIP_INTREG	Mapped to SIP register within TOE100G-IP
BA+0x0018	TOE_DPN_INTREG	Mapped to DPN register within TOE100G-IP
BA+0x001C	TOE_SPN_INTREG	Mapped to SPN register within TOE100G-IP
BA+0x0020	TOE_TDL_INTREG	Mapped to TDL register within TOE100G-IP
BA+0x0024	TOE_TMO_INTREG	Mapped to TMO register within TOE100G-IP
BA+0x0028	TOE_PKL_INTREG	Mapped to PKL register within TOE100G-IP
BA+0x002C	TOE_PSH_INTREG	Mapped to PSH register within TOE100G-IP
BA+0x0030	TOE_WIN_INTREG	Mapped to WIN register within TOE100G-IP
BA+0x0034	TOE_ETL_INTREG	Mapped to ETL register within TOE100G-IP
BA+0x0038	TOE_SRV_INTREG	Mapped to SRV register within TOE100G-IP
BA+0x003C	TOE_VER_INTREG	Mapped to VER register within TOE100G-IP
BA+0x0040	TOE_DML_INTREG	Mapped to DML register within TOE100G-IP
BA+0x0044	TOE_DMH_INTREG	Mapped to DMH register within TOE100G-IP
BA+0x1000 – BA+0x10FF: UserReg control/status		
BA+0x1000 Wr/Rd	Total transmit length (USER_TXLEN_INTREG)	Wr [31:0] – Total amount of transmitted data in 512-bit unit. Valid from 1-0xFFFFFFFF. Rd [31:0] – Current amount of transmitted data in 512-bit unit. The value is cleared to 0 when USER_CMD_INTREG is written by user.
BA+0x1004 Wr/Rd	User Command (USER_CMD_INTREG)	Wr [0] – Start transmitting. Set '0' to start transmitting data. [1] – Data verification enable ('0': Disable data verification, '1': Enable data verification) Rd [0] – Busy of PattGen inside UserReg ('0': Idle, '1': PattGen is busy) [1] – Data verification error ('0': Normal, '1': Error) This bit is auto-cleared when user starts new operation or reset. [2] – Mapped to ConnOn signal of TOE100G-IP
BA+0x1008 Wr/Rd	User Reset (USER_RST_INTREG)	Wr [0] – Reset signal. Set '1' to reset UserReg. This bit is auto-cleared to '0'. [8] – Set '1' to clear TimerInt latched value Rd [8] – Latched value of TimerInt output from IP ('0': Normal, '1': TimerInt='1' is detected) This flag can be cleared by system reset condition or setting USER_RST_INTREG[8]='1'.
BA+0x100C Rd	FIFO status (USER_FFSTS_INTREG)	Rd[5:0] - Mapped to TCPRxFfLastRdCnt signal of TOE100G-IP [15:6] - Mapped to TCPRxFfRdCnt signal of TOE100G-IP [24] - Mapped to TCPTxFfFull signal of TOE100G-IP
BA+0x1010 Rd	Total receive length (USER_RXLEN_INTREG)	Rd[31:0] – Current amount of received data in 512-bit unit The value is cleared to 0 when USER_CMD_INTREG is written by user.

Address Wr/Rd	Register Name (Label in the "ping100gtest.c" and "dhcp100gtest.c")	Description
BA+0x2000 – BA+0x3FFF: UserTxMAC		
BA+0x2000 Wr/Rd	UserTxMAC transmit length (TXEMAC_LEN_INTREG)	Wr [11:0] – Total amount of transmitted data in byte unit. Valid from 1 – 4095. UserTxMAC starts transmitting packet to EMAC after this register is set. Rd [0] – Busy flag of UserTxMAC ('0': Idle, '1': Busy)
BA+0x3000- BA+0x3FFF Wr	TxRAM in UserTxMAC (TXRAM_BASE_ADDR)	TxRAM for storing transmitted packet that is created by CPU for slow-port connection
BA+0x4000 – BA+0x5FFF: UserRxMAC		
BA+0x4000 – BA+0x4027 Wr	UserRxMAC header data (RXEMAC_HDVAL_ADDR)	38-byte header data set to packet filtering inside UserRxMAC for comparing byte#0-byte#37 of the received packet. To start UserRxMAC operation, RXEMAC_CMD_INTREG[0] must be set to '1' (enable received packet) *0x4000[7:0], [15:8], [23:16], [31:24] – byte#0, #1, #2, #3* *0x4004[7:0], [15:8], [23:16], [31:24] – byte#4, #5, #6, #7* *…* *0x4020[7:0], [15:8], [23:16], [31:24] – byte#32, #33, #34* *0x4024[7:0], [15:8] – byte#36, #37*
BA+0x4040 – BA+0x4047 Wr	UserRxMAC header byte enable (RXEMAC_HDEN_ADDR)	Byte enable to verify 38-byte header data. One bit is referred to one byte. 0x4040[0], [1], [2], …, [31] – Enable of byte#0, #1, #2, …, #31 0x4044[0], [1], [2], …, [5] – Enable of byte#32, #33, #34, …, #37 '0': Disable byte filtering (Bypass), '1': Enable byte filtering
BA+0x4060 Wr	UserRxMAC Command (RXEMAC_CMD_INTREG)	[0] – UserRxMAC enable '0': Disable UserRxMAC, '1': Enable UserRxMAC [1] – UserRxMAC FIFO read enable User sets this bit to '1' after finishing reading data from RxMacFf (RXEMAC_FF_INTREG[8:0]). The FIFO read enable is asserted to '1' for one cycle when this bit is written to '1' by user.
BA+0x4064 Rd	UserRxMAC FIFO (RXEMAC_FF_INTREG)	[5:0] – Read data of RxMacFf [15] – Empty flag of RxMacFf
BA+0x5000 – BA+0x5FFF Rd	RxRAM in UserRxMAC RXRAM_BASE_ADDR	RxRAM for storing received packet that has the valid header. CPU reads received packet for slow-port connection processing
BA+0x6000 – BA+0x6FFF: Ethernet MAC		
BA+0x6000 Rd	EMAC IP version (EMAC_VER_INTREG)	[31:0] – Mapped to IPVersion output from DG EMAC-IP when the system integrates DG EMAC-IP. In this demo, it is equal to 0.
BA+0x6004 Rd	EMAC Status (EMAC_STS_INTREG)	[0] – Ethernet linkup status from Ethernet MAC ('0': Not linkup, '1': Linkup)

Pattern Generator

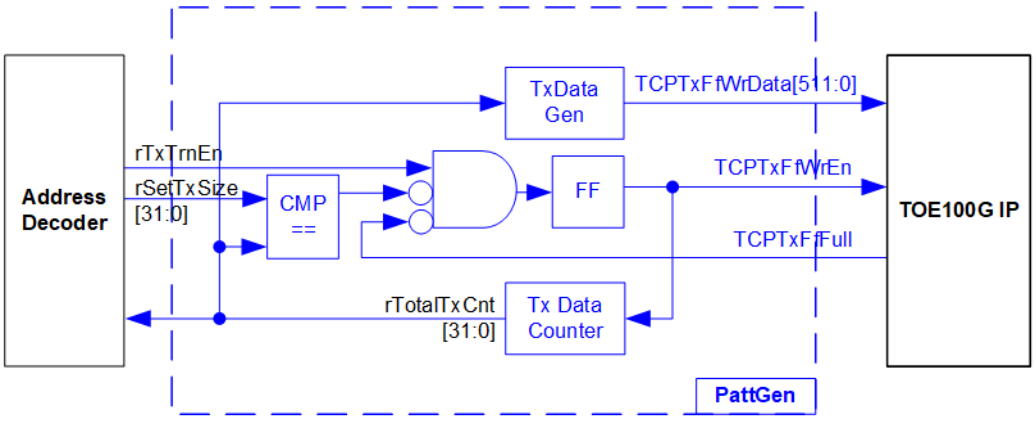

Figure 2-13 PattGen block

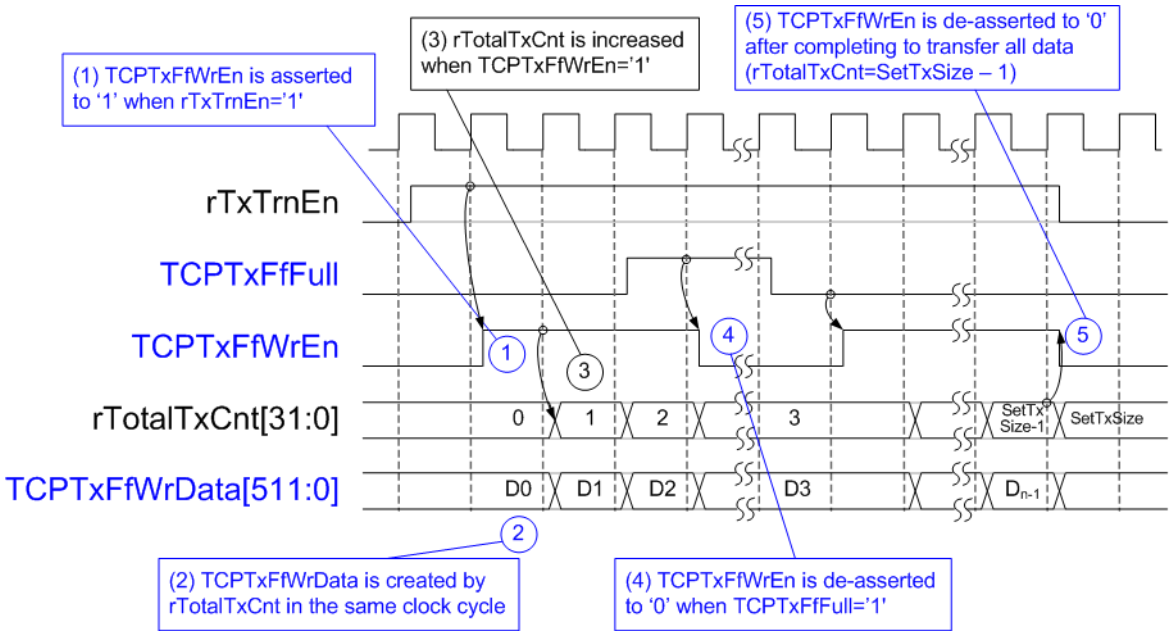

Figure 2-14 PattGen timing diagram

Figure 2-13 shows the details of PattGen which generates test data to TOE100G-IP. Timing diagram to show the relation of each logic is displayed in Figure 2-14.

To start PattGen operation, the user sets USER_CMD_INTREG[0]='0' and then rTxTrnEn is asserted to '1'. When rTxTrnEn is '1', TCPTxFfWrEn is controlled by TCPTxFfFull. TCPTxFfWrEn is de-asserted to '0' when TCPTxFfFull is '1'. rTotalTxCnt is the data counter to check total amount of transmitted data to TOE100G-IP. Also, rTotalTxCnt is used to generate 32-bit incremental data for TCPTxFfWrData signal. After all data is transferred completely (Total amount of data is equal to rSetTxSize-1), rTxTrnEn is de-asserted to '0'.

Pattern Verification

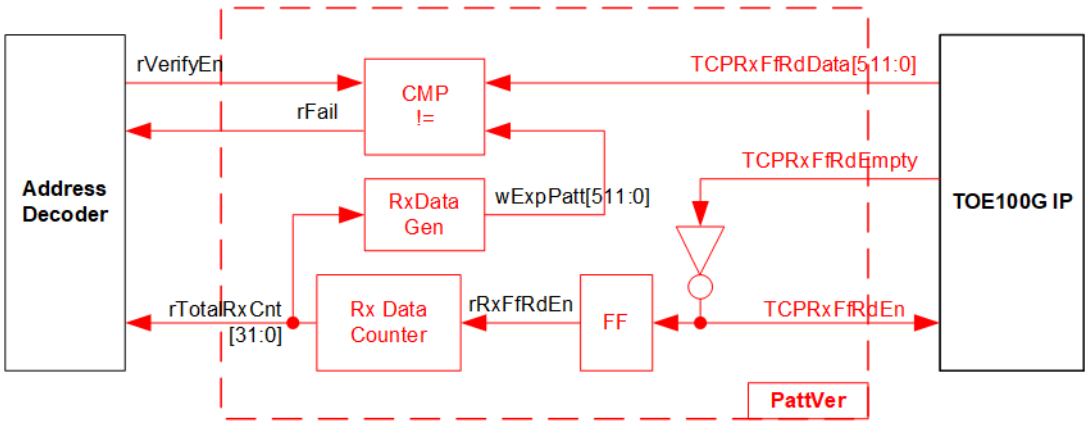

Figure 2-15 PattVer block

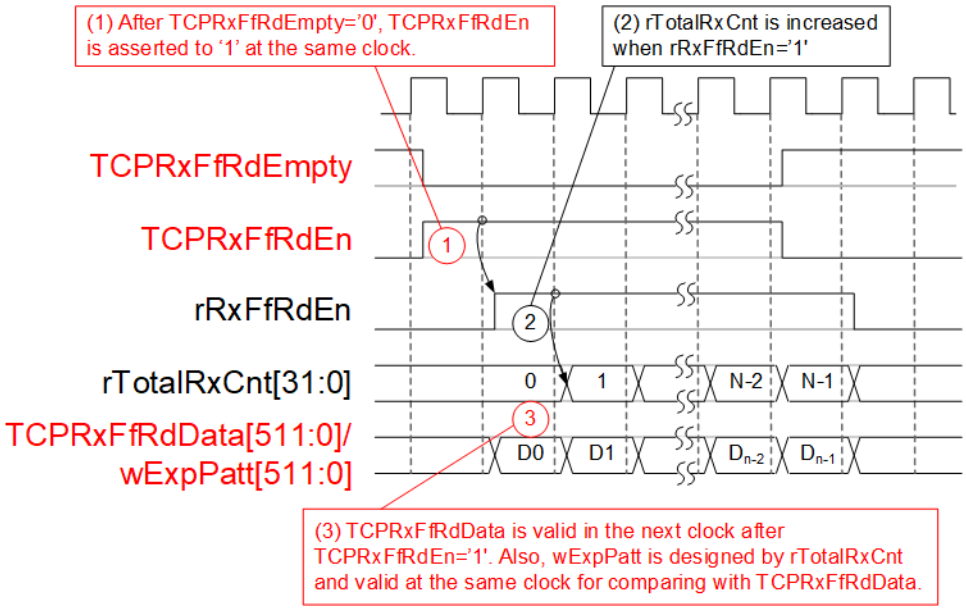

Figure 2-16 PattVer Timing diagram

Figure 2-15 shows the details of PattVer logic for reading the data from TOE100G-IP with or without data verification, controlled by rVerifyEn flag which is set by the user. Timing diagram of the logic is displayed in Figure 2-16.

When rVerifyEn is set to '1', data comparison is enabled to compare read data (TCPRxFfRdData) with the expected pattern (wExpPatt). If data verification is failed, rFail is asserted to '1'. TCPRxFfRdEn is designed by using NOT logic of TCPRxFfRdEmpty. TCPRxFfRdData is valid for data comparison in the next clock. rRxFfRdEn, one clock latency of TCPRxFfRdEn, is applied to be counter enable of rTotalRxCnt, counting total amount of received data. rTotalRxCnt is used to generate wExpPatt, so wExpPatt is valid at the same time as TCPRxFfRdData valid.

3 CPU firmware and Test software of Ping demo

The Ping firmware is modified from the standard TOE100G-IP which supports one fast-port connection to support slow-port operation. The ICMP protocol is implemented to generate ICMP Echo reply after receiving ICMP Echo request. More details of Ping demo are described as follows.

Note: The standard reference design document can be downloaded by following website.
https://dgway.com/products/IP/TOE100G-IP/dg_toe100gip_cpu_refdesign_xilinx.pdf

3.1 Firmware sequence

After FPGA boot-up, 100G Ethernet link status (EMAC_STS_INTREG[0]) is polling. The CPU waits until Ethernet link is established. Next, welcome message is displayed and user selects the initialization mode of TOE100G-IP to be Client, Server, or Fixed-MAC. To run the test with PC, it is recommended to set initialization mode to be Client mode to get the MAC address of the target device by sending ARP request. After that, the default parameters are displayed on the console. User enters the keys to start the initialization by using default parameters or updated parameters, as shown in Figure 3-1.

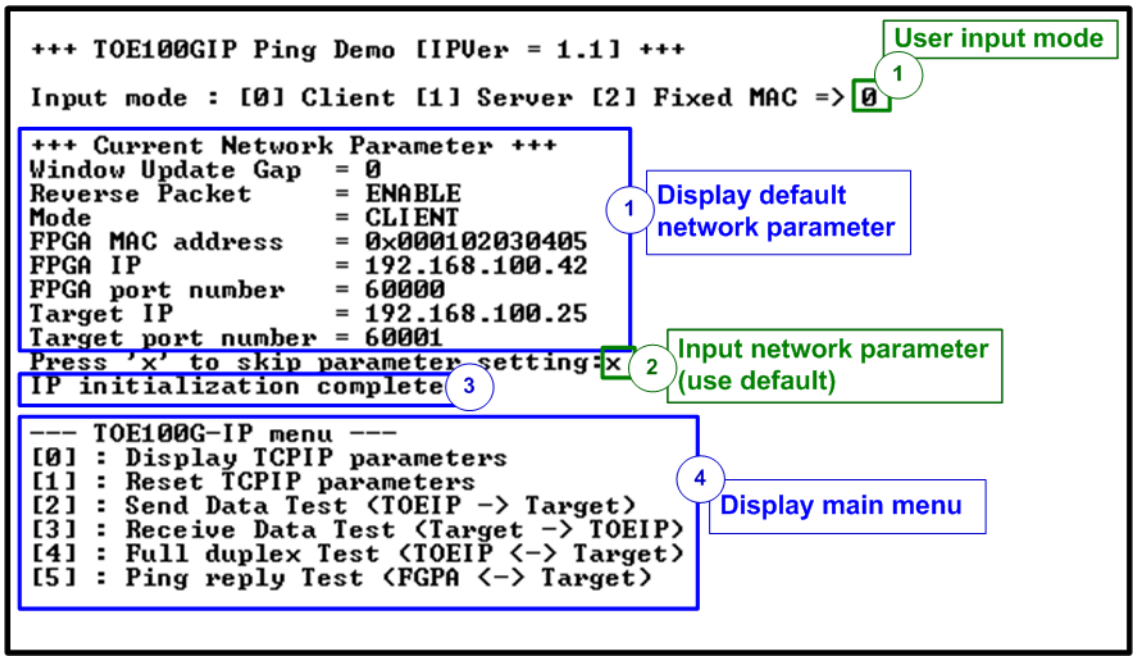

Figure 3-1 Boot menu of Ping demo

There are four steps to complete initialization process, described as follows.

1) CPU receives the initialization mode and then displays default parameters of the selected mode on the console.
2) User inputs 'x' to complete initialization process by using default parameters. Other keys are set for changing some parameters. More details for changing some parameters are described in Reset IP menu (topic 3.1.2).
3) CPU waits until TOE100G-IP finishes initialization process (TOE_CMD_INTREG[0]='0').
4) Main menu is displayed. There are six test operations for user selection. More details of each menu are described as follows.

3.1.1 Display parameters

This menu is used to show current parameters of TOE100G-IP, i.e., the initialization mode, Windows update threshold, Reverse packet enable, source MAC address, destination IP address, source IP address, destination port, source port, and destination MAC address (when using Fixed MAC mode). The sequence of display parameters is as follows.

1) Read all network parameters from each variable in firmware.
2) Print out each variable.

3.1.2 Reset IP

This menu is used to change TOE100G-IP parameters such as IP address and source port number. After setting updated parameter to TOE100G-IP register, the CPU resets the IP to re-initialize by using new parameters. Finally, the CPU monitors busy flag to wait until the initialization is completed. The sequence to reset IP is as follows.

1) Display current parameter value to the console.
2) Receive initialization mode from user and confirm that the input is valid. If initialization mode is changed, the latest parameter set of new mode is displayed on the console.
3) Receive remaining input parameters from user and check if input is valid or not. When the input is invalid, the parameter is not updated.
4) Reset PattGen and PattVer logic by sending reset to user logic (USER_RST_INTREG[0]='1').
5) Force reset to IP by setting TOE_RST_INTREG[0]='1'.
6) Set all parameters to TOE100G-IP register such as TOE_SML_INTREG and TOE_DIP_INTREG.
7) De-assert IP reset by setting TOE_RST_INTREG[0]='0'. After that, TOE100G-IP starts the initialization process.
8) Monitor IP busy flag (TOE_CMD_INTREG[0]) until the initialization process is completed (busy flag is de-asserted to '0').

3.1.3 Send data test

Three user inputs are received to set total transmit length, packet size, and connection mode (active open for client operation or passive open for server operation). The operation is cancelled if some inputs are invalid. During running the test, 32-bit incremental data is generated from the logic and sent to PC. Data is verified by Test application on PC. The operation is finished when total data are transferred from FPGA to PC. The sequence of the test is as follows.

1) Receive transfer size, packet size, and connection mode from user and verify if all inputs are valid.
2) Set UserReg registers, i.e., transfer size (USER_TXLEN_INTREG), reset flag to clear initial value of test pattern (USER_RST_INTREG[0]='1'), and command register to start data pattern generator (USER_CMD_INTREG=0). After that, test pattern generator in UserReg starts sending data to TOE100G-IP.
3) Display recommended parameters of test application on PC by reading current system parameters.
4) Open connection following connection mode setting.
   i) For active open, CPU sets TOE_CMD_INTREG=2 (Open port) and waits until ConnOn status (USER_CMD_INTREG[2]) is equal to '1'.
   ii) For passive open, CPU waits until connection is opened by another device (PC). ConnOn status (USER_CMD_INTREG[2]) is monitored until it is equal to '1'.
5) Set packet size to TOE100G-IP register (TOE_PKL_INTREG) and calculate total number of loops from total transfer size. Maximum transfer size of each loop is 4 GB. The operation of each loop is as follows.
   i) Set transfer size of this loop to TOE100G-IP register (TOE_TDL_INTREG). Transfer size is fixed to 4 GB except the last loop which is equal to the remaining size.
   ii) Set send command to TOE100G-IP register (TOE_CMD_INTREG=0).
   iii) Wait until operation is completed by monitoring busy flag (TOE_CMD_INTREG[0]='0'). During monitoring busy flag, CPU reads current amount of transmitted data from user logic (USER_TXLEN_INTREG) and displays the results on the console every second.
6) Set close connection command to TOE100G-IP register (TOE_CMD_INTREG=3).
7) Calculate performance and show test result on the console.

3.1.4 Receive data test

User sets total amount of received data, data verification mode (enable or disable), and connection mode (active open for client operation or passive open for server operation). The operation is cancelled if some inputs are invalid. During running the test, 32-bit incremental data is generated to verify the received data from another device (PC) when data verification mode is enabled. The sequence of this test is as follows.

1) Receive total transfer size, data verification mode, and connection mode from user input. Verify that all inputs are valid.
2) Set UserReg registers, i.e., reset flag to clear the initial value of test pattern (USER_RST_INTREG[0]='1') and data verification mode (USER_CMD_INTREG[1]='0' or '1').
3) Display recommended parameter (similar to Step 3 of Send data test).
4) Open connection following connection mode (similar to Step 4 of Send data test).
5) Wait until connection is closed by another device (PC). Connon status (USER_CMD_INTREG[2]) is monitored until it is equal to '0'. During monitoring Connon, CPU reads current amount of received data from user logic (USER_RXLEN_INTREG) and displays the results on the console every second.
6) Wait until all data are read by user logic completely by checking FIFO status (USER_FFSTS_INTREG[15:0]=0).
7) Compare total amount of received data in user logic (USER_RXLEN_INTREG) with the set value. If all data is completely received, CPU checks verification result by reading USER_CMD_INTREG[1] ('0': normal, '1': error). If some errors are detected, the error message is displayed.
8) Calculate performance and show test result on the console.

3.1.5 Full duplex test

This menu is designed to run full duplex test by transferring data between FPGA and another device (PC) in both directions by using the same port number at the same time. Four inputs are received from user, i.e., total data size for both transfer directions, packet size for FPGA sending logic, data verification mode for FPGA receiving logic, and connection mode (active open/close for client operation or passive open/close for server operation).

The transfer size set on FPGA must be matched to the size set on test application (tcp_client_txrx_40G). Connection mode on FPGA must be set to passive (server operation).

The test runs in forever loop until the user cancels operation. The operation can be cancelled by entering any keys on FPGA console and then entering Ctrl+C on PC console. The sequence of this test is as follows.

1) Receive total data size, packet size, data verification mode, and connection mode from the user and verify that all inputs are valid.
2) Display the recommended parameters of test application run on PC from the current system parameters.
3) Set UserReg registers, i.e., transfer size (USER_TXLEN_INTREG), reset flag to clear the initial value of test pattern (USER_RST_INTREG[0]='1'), and command register to start data pattern generator with data verification mode (USER_CMD_INTREG=1 or 3).
4) Open connection following the connection mode value (similar to Step 4 of Send data test).
5) Set packet size to TOE100G-IP registers (TOE_PKL_INTREG=user input) and calculate total transfer size in each loop. Maximum size of one loop is 4 GB. The operation of each loop is as follows.
   i) Set transfer size of this loop to TOE_TDL_INTREG. Transfer size is fixed to maximum size (4GB) which is also aligned to packet size, except the last loop. The transfer size of the last loop is equal to the remaining size.
   ii) Set send command to TOE100G-IP register (TOE_CMD_INTREG=0).
   iii) Wait until send command is completed by monitoring busy flag (TOE_CMD_INTREG[0] ='0'). During monitoring busy flag, CPU reads current amount of transmitted data and received data from user logic (USER_TXLEN_INTREG and USER_RXLEN_INTREG) and displays the results on the console every second.
6) Close connection following connection mode value.
   a. For active close, CPU waits until total amount of received data is equal to the set value from user. Then, set USER_CMD_INTREG=3 to close connection. Next, CPU waits until connection is closed by monitoring ConnOn (USER_CMD_INTREG[2]='0').
   b. For passive close, CPU waits until the connection is closed by another device (PC or FPGA). Connon status (USER_CMD_INTREG[2]) is monitored until it is equal to '0'.
7) Check the result and the error (similar to Step 6-7 of Receive data test).
8) Calculate performance and show the test result on the console. Go back to step 3 to repeat the test in forever loop.

3.1.6 Ping reply test

When Test PC runs Ping command to check round-trip time, ICMP Echo request packet is created by Test PC and then it waits until ICMP Echo reply is received to measure the latency time. Therefore, this menu is designed to configure the hardware to receive ICMP Echo request. If the request packet is valid, ICMP Echo reply is created to be the response packet. Packet structure of ICMP protocol for echo request/reply type is shown in Figure 3-2.

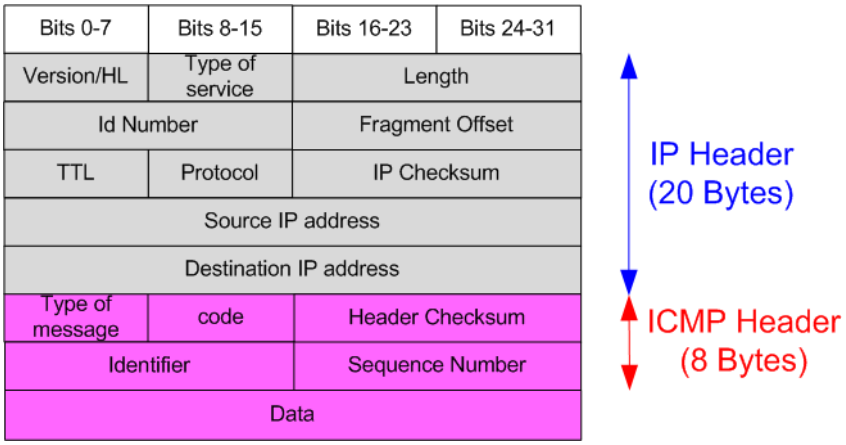

Figure 3-2 Packet structure for ICMP request/reply packet

The type value of the Echo request packet is equal to 8 while the type value of the Echo reply is equal to 0. More information about Ping command is described from following website.

http://en.wikipedia.org/wiki/Ping_(networking_utility)

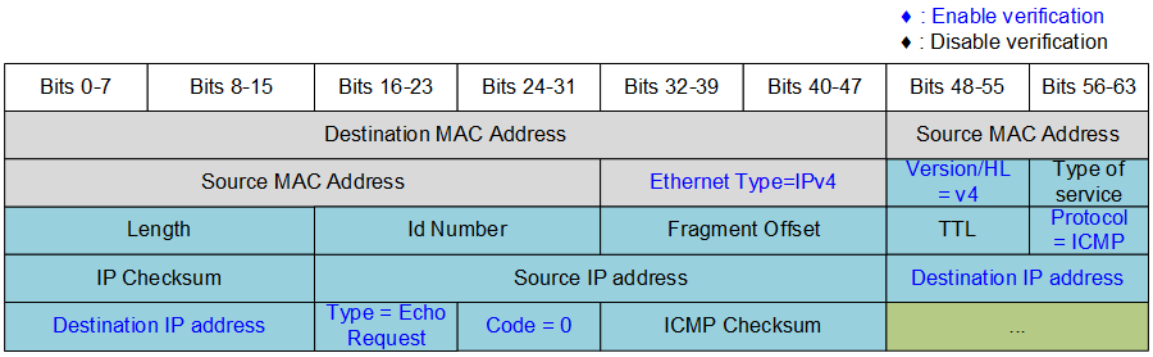

Figure 3-3 ICMP Echo request packet filtering

The sequence to run Ping reply is described as follows.

1) Set the filtering parameters of UserRxMAC to receive only ICMP Echo request packet by using init_filter function, as described in topic 3.2. The filtering data of ICMP Echo request packet contains the following fields (blue color in Figure 3-3).
   - Ethernet Type (2 bytes) = 0x0800 (IPv4)
   - IP version (1 byte) = 0x45 (Version 4)
   - Protocol (1 byte) = 0x01 (ICMP Protocol)
   - Destination IP Address (4 bytes) = IP address of FPGA
   - ICMP type (1 byte) = 0x08 (Echo Request)
   - ICMP code (1 bytes) = 0x00 (Echo Request code)

   Note: Figure 3-3 shows only 38-byte of ICMP packet, the actual size of ICMP packet is 42 bytes (Rest of header field is excluded) because the filtering logic in UserRxMAC module is designed to support up to 38-byte header data.
2) Enable receive data mode of UserRxMac module (RXEMAC_CMD_INTREG[0]='1').
3) Wait until there is new packet stored in RxRAM by checking empty flag of RxMacFifo (RXEMAC_FF_INTREG[15]='0').
4) Read and validate the last address of the received packet from RxMacFifo (RXEMAC_FF_INTREG[5:0]). After that, asserts read acknowledge to flush the current read data from RxMacFifo (RXEMAC_CMD_INTREG[1]='1').
5) Copy the received packet from RxRAM (RXRAM_BASE_ADDR) to receive temporal buffer (rxbuff_ch) in the firmware.
6) Decode the received packet. Continue the next step if the packet is Echo request packet and the parameters and checksum are correct. Otherwise, error message is displayed.
7) Prepare Echo reply packet in transmit temporal buffer (txbuff_ch) in the firmware. IP checksum and ICMP checksum are calculated by CPU to be a part of Echo reply packet header. After that, copy data from transmit temporal buffer (txbuff_ch) to TxRAM (TXRAM_BASE_ADDR).
8) Set UserTxEMAC register to start data sending by setting TXMAC_LEN_INTREG=Echo reply packet length.
9) Go back to step 2 to run the test in forever loop until the user cancel operation by pressing any key on console.
10) Disable receive data mode (RXEMAC_CMD_INTREG[0]='0') before returning to main menu. Therefore, the slow-port connection is not operated until this menu is re-selected.

3.2 Function list in User application

This topic describes the function list to run the Ping firmware that can be divided into two groups. The first group is the function list for operating TOE100G-IP and the second is the function list for running the Ping test.

3.2.1 Function for operating fast-port connection by TOE100G-IP

void exec_port(unsigned int port_ctl, unsigned int mode_active)	
Parameters	port_ctl: 1-Open port, 0-Close port mode_active: 1-Active open/close, 0-Passive open/close
Return value	None
Description	For active mode, write TOE_CMD_INTREG to open or close connection. After that, call read_conon function to monitor connection status until it changes from ON to OFF or OFF to ON, depending on port_ctl mode.

void init_param(void)	
Parameters	None
Return value	None
Description	Ask the user if the parameters are updated. After that, set the parameters to TOE100G-IP registers from global parameters. After reset is de-asserted, it waits until TOE100G-IP busy flag is de-asserted to '0'.

int input_param(void)	
Parameters	None
Return value	0: Valid input, -1: Invalid input
Description	Receive network parameters from user, i.e., Mode, Reverse packet enable, Window threshold, FPGA MAC address, FPGA IP address, FPGA port number, Target IP address, Target port number, and Target MAC address (when run in Fixed MAC mode). If the input is valid, the parameter is updated. Otherwise, the value does not change. After receiving all parameters, the current value of all parameter is displayed.

unsigned int read_conon(void)	
Parameters	None
Return value	0: Connection is OFF, 1: Connection is ON.
Description	Read value from USER_CMD_INTREG register and return only bit2 value to show connection status.

void show_cursize(void)	
Parameters	None
Return value	None
Description	Read USER_TXLEN_INTREG and USER_RXLEN_INTREG and then display the current amount of transmitted data and received data in Byte, KByte, or MByte unit.

void show_param(void)	
Parameters	None
Return value	None
Description	Display the current value of the network parameters set to TOE100G-IP such as IP address, MAC address, and port number.

void show_result(void)	
Parameters	None
Return value	None
Description	Read USER_TXLEN_INTREG and USER_RXLEN_INTREG to display total amount of transmitted data and received data. Next, read the global parameters (timer_val and timer_upper_val) and calculate total time usage to display in usec, msec, or sec unit. Finally, transfer performance is calculated and displayed in MB/s unit.

int toe_recv_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Receive data test following description in topic 3.1.4

int toe_send_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Send data test following description in topic 3.1.3

int toe_txrx_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Full duplex test following described in topic 3.1.5

void wait_ethlink(void)	
Parameters	None
Return value	None
Description	Read USER_RST_INTREG[16] and wait until ethernet connection is established.

3.2.2 Function for operating slow-port connection by CPU (Ping)

unsigned int cal_checksum (unsigned int byte_len, unsigned char *buf)	
Parameters	byte_len: The length of data in byte unit buf: Pointer to the first byte data position
Return value	16-bit checksum of the data
Description	Calculate the 16-bit checksum value of the data. User must prepare the array of data in character data type and calculate the length of data. Then calls the function by using the length of data and character pointer to the first data in array. The return value is the calculated 16-bit checksum value of the data.

void init_filter (void)	
Parameters	None
Return value	None
Description	Write RXEMAC_CMD_INTREG to pause the storing of received packet in the slow-port connection. Next, Write RXEMAC_HDVAL_ADDR and RXEMAC_HDEN_ADDR for filtering an ICMP Echo request packet as shown in blue color text of Figure 3-3.

void ping_test (void)	
Parameters	None
Return value	None
Description	Run Ping reply test following description in topic 3.1.6

unsigned int prepare_rxbuffer (void)	
Parameters	None
Return value	The length of received packet in byte unit. Return 0 if there is no received packet.
Description	Read RXEMAC_FF_INTREG to check whether there is any received data packet in the slow-port connection or not. If received packet is stored hardware, read the packet length from RxMacFifo by reading at RXEMAC_FF_INTREG[5:0]. After that, flush the current data from RxMacFifo by writing RXEMAC_CMD_INTREG[1]='1'. Next, copy the data from the hardware (RXRAM_BASE_ADDR) to the character array variable in CPU. Finally, return the read length in byte unit.

3.3 Test Software on PC

3.3.1 Tcpdatatest for half duplex

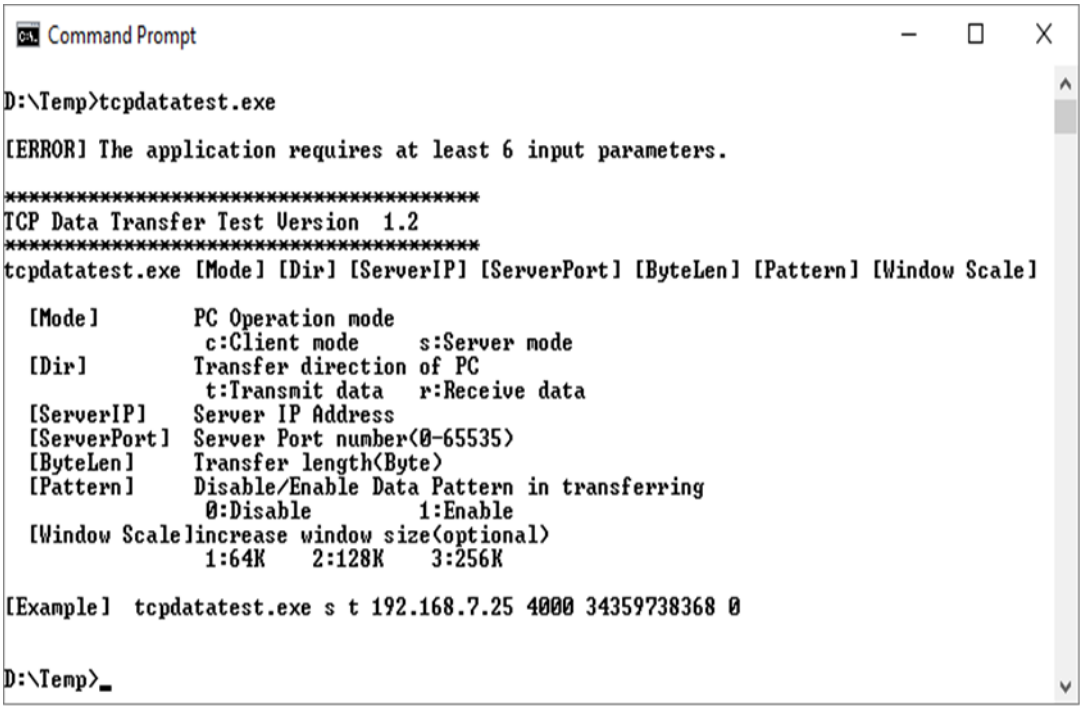

Figure 3-4 “tcpdatatest” application usage

“tcpdatatest” is designed to run on PC for sending or receiving TCP data via Ethernet as server or client mode. PC of this demo should run in client mode. User sets parameters to select transfer direction and the mode. Six parameters are required as follows.

1) Mode: c –PC runs in Client mode and FPGA runs in Server mode
2) Dir: t – transmit mode (PC sends data to FPGA)
   r – receive mode (PC receives data from FPGA)
3) ServerIP: IP address of FPGA when PC runs in Client mode (default is 192.168.100.42)
4) ServerPort: Port number of FPGA when PC runs in Client mode (default is 4000)
5) ByteLen: Total transfer size in byte unit. This input is used in transmit mode only and ignored in receive mode. In receive mode, the application is closed when the connection is terminated. In transmit mode, ByteLen must be equal to the total transfer size, set in receive data test menu of FPGA.
6) Pattern:
   0 – Generate dummy data in transmit mode or disable data verification in receive mode.
   1 – Generate incremental data in transmit mode or enable data verification in receive mode.

Note: Window Scale is the optional parameter which is not used in the demo.

Transmit data mode

Following sequence is the sequence when test application runs in transmit mode.

1) Get parameters from the user and verify that all inputs are valid.
2) Create the socket and set socket options.
3) Create the new connection by using server IP address and server port number.
4) Allocate 1 MB memory to be send buffer.
5) Skip this step if the dummy pattern is selected. Otherwise, generate the incremental test pattern to send buffer.
6) Send data out and read total sent data from the function.
7) Calculate remaining transfer size.
8) Print total transfer size every second.
9) Repeat step 5) – 8) until the remaining transfer size is 0.
10) Calculate total performance and print the result on the console.
11) Close the socket and free the memory.

Receive data mode

Following sequence is the sequence when test application runs in receive mode.

1) Follow the step 1) – 3) of Transmit data mode.
2) Allocate memory to be receive buffer.
3) Read data from the receive buffer and increase total amount of received data.
4) This step is skipped if data verification is disabled. Otherwise, received data is verified by the incremental pattern. Error message is printed out when data is not correct.
5) Print total amount of received data every second.
6) Repeat step 3) – 5) until the connection is closed.
7) Calculate total performance and print the result on the console.
8) Close socket and free the memory.

3.3.2 tcp_client_txrx_40G" application

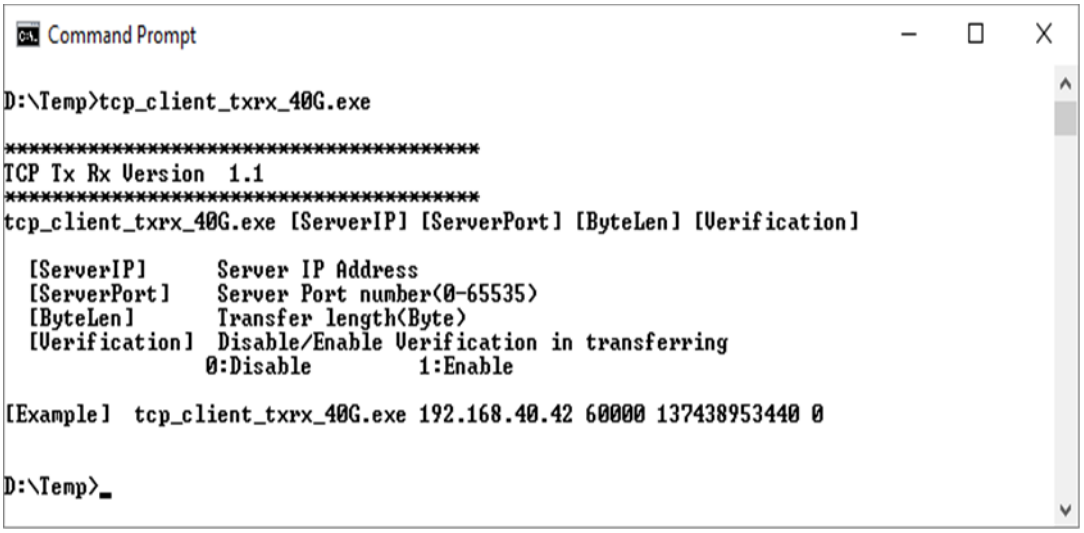

Figure 3-5 "tcp_client_txrx_40G" application usage

"tcp_client_txrx_40G" application is designed to run on PC for sending and receiving TCP data through Ethernet by using the same port number at the same time. The application is run in Client mode, so user needs to input server parameters (the network parameters of TOE100G-IP). As shown in Figure 3-5, there are four parameters to run the application, described as follow.

1) ServerIP : IP address of FPGA
2) ServerPort : Port number of FPGA
3) ByteLen : Total transfer size in byte unit. This is total amount of transmitted data and received data. This value must be equal to the transfer size set on FPGA for running full-duplex test.
4) Verification :

0 – Generate dummy data for sending function and disable data verification for receiving function. This mode is used to check the best performance of full-duplex transfer.

1 – Generate incremental data for sending function and enable data verification for receiving function.

The sequence of test application is as follows.

1) Get parameters from the user and verify that the input is valid.
2) Create the socket and set socket options.
3) Create the new connection by using server IP address and server port number.
4) Allocate 64 KB memory for send and receive buffer.
5) Generate incremental test pattern to send buffer when the test pattern is enabled. Skip this step if dummy pattern is selected.
6) Send data out, read total send data from the function, and calculate remaining send size.
7) Read data from the receive buffer and increase total amount of received data.
8) Skip this step if data verification is disabled. Otherwise, data is verified by incremental pattern. Error message is printed out when data is not correct.
9) Print total amount of transmitted data and received data every second.
10) Repeat step 5) – 9) until total amount of transmitted data and received data are equal to ByteLen, set by user.
11) Calculate performance and print the result on the console.
12) Close the socket.
13) Sleep for 1 second to wait until the hardware completes the current test loop.
14) Run step 3) – 13) in forever loop. If verification is failed, the application is stopped.

4 CPU Firmware and Test software of DHCP demo

Similar to the Ping firmware, the DHCP firmware is also modified from the standard TOE100G-IP to operate the slow-port operation. The slow-port connection implements to support the DHCP operation for dynamic IP address assignment (by DHCP server). More details of the DHCP firmware are described as follows.

4.1 Firmware sequence

The boot-up sequence of DHCP test is different from the standard reference design and the Ping firmware because the IP address is assigned by DHCP server (run on Test PC), not by user. Thus, DHCP operation is run to get IP address before starting TOE100G-IP initialization.

After FPGA boot-up, 100G Ethernet link status (EMAC_STS_INTREG[0]) is polling. The CPU waits until Ethernet link is established. Next, the recommended DHCP parameters is displayed and user selects to use the recommended DHCP parameters or to change the parameters. The completion message of DHCP operation is displayed after the firmware obtains the IP address by DHCP operation. Next, the message to start the IP initialization is displayed and user selects the initialization mode of TOE100G-IP to be Client, Server, or Fixed-MAC. To run the test with PC, it is recommended to set initialization mode to be Client mode to get the MAC address of the target device by sending ARP request. After that, the default parameters for TOE100G-IP initialization are displayed on the console. User enters the keys to start TOE100G-IP initialization by using default parameters or updated parameters, as shown in Figure 4-1.

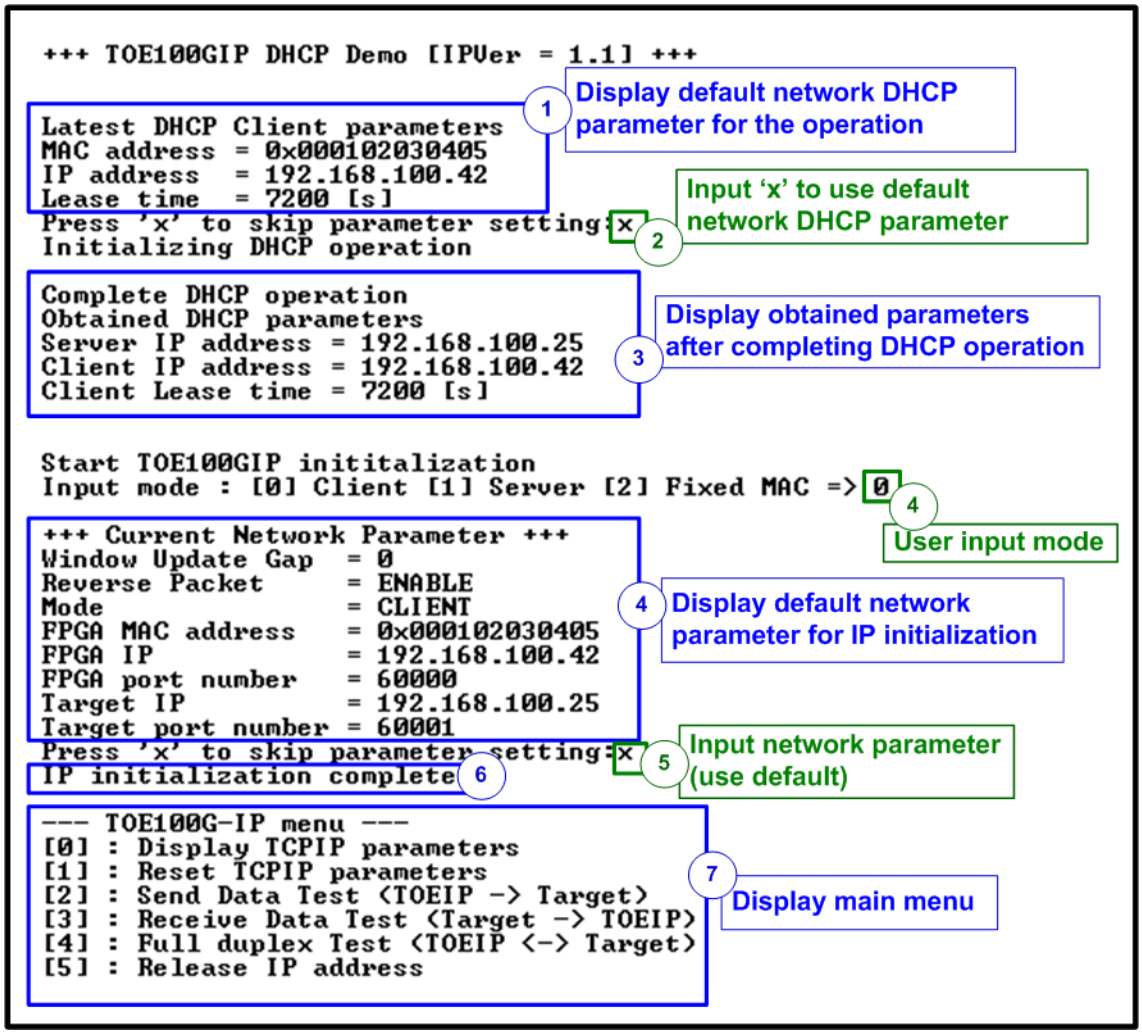

Figure 4-1 Boot menu of DHCP demo

There are seven steps to complete initialization process as follows.

1) Recommended DHCP parameters is displayed after booting the system.
2) User inputs 'x' to initial DHCP operation by using default parameters. Other keys are set for changing some parameters. More details for changing some parameters are described in Release IP address menu.
3) CPU completes the DHCP operation and displays the obtained parameters from server.
4) CPU receives the initialization mode and then displays default parameters of the selected mode on the console.
5) User inputs 'x' to complete IP initialization process by using default parameters. Other keys are set for changing some parameters. More details for changing some parameters are described in Reset IP menu.
6) CPU waits until TOE100G-IP finishes initialization process (TOE_CMD_INTREG[0]='0').
7) Main menu is displayed. There are five test operations for user selection.

The test menus for running the fast port connection (menu[0] – menu[4]) have the same operation as Ping demo. Please see more details from topic 3.1.1 - 3.1.5. However, there is slightly modified in Reset IP menu to exclude the assignment of FPGA IP address and FPGA MAC address. Both parameters are assigned by DHCP server in this test. Therefore, only Release IP address menu is described in more details as follows.

DHCP overview

After this menu is selected, the IP address that is assigned by DHCP will be released. After that, the boot menu to start IP address assignment by DHCP is rerun. Therefore, this menu is implemented for two functions, obtaining the IP address and releasing the IP address by using slow-port connection to transmit and receive DHCP packet. The operation is handled by CPU.

There are two DHCP packets to obtain the IP address, i.e., DHCP discover packet and DHCP request packet. While there is only one DHCP packet transmitted to release the IP address - DHCP release packet. Thus, CPU firmware needs to generate these DHCP packets. On the contrary, there are three types of DHCP packet that implements in receiving path for DHCP operation, i.e., DHCP offer, DHCP ACK, and DHCP NAK. Packet structure of DHCP protocol is shown in Figure 4-2.

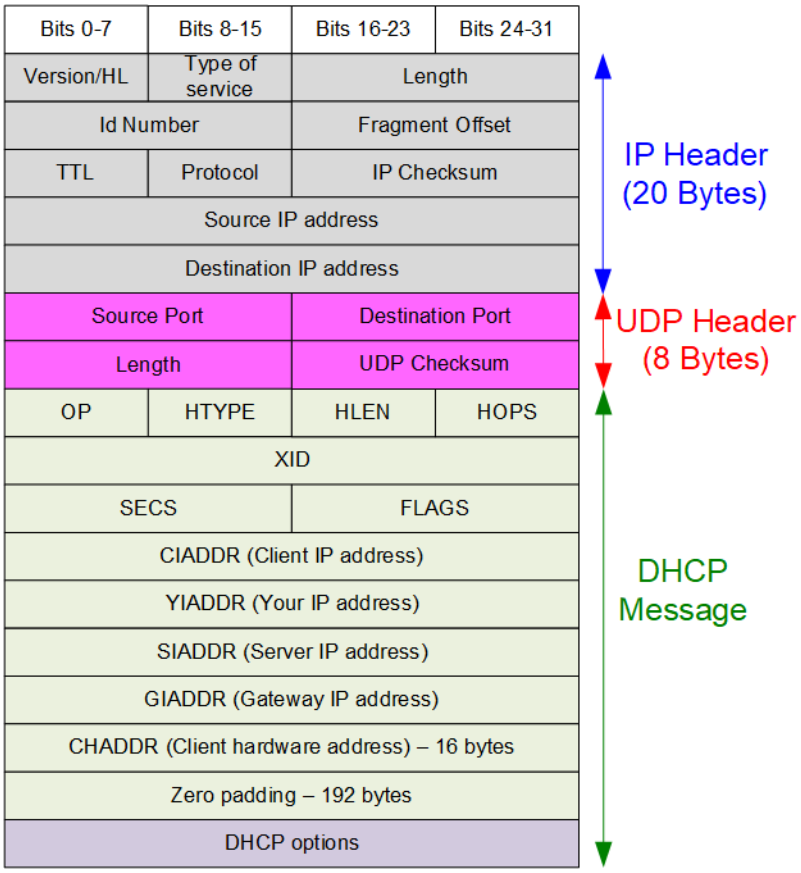

Figure 4-2 DHCP packet structure

The DHCP operation uses the UDP over IP protocol in order to communicate with other devices. The operation to request IP address from the server is shown in Figure 4-3.

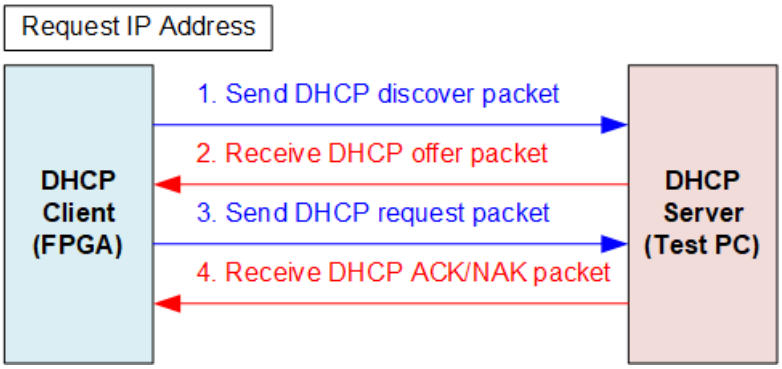

Figure 4-3 Request IP address by DHCP

The operation begins with the client device (FPGA) broadcasting the DHCP discover packet. After that, any DHCP servers (PC) that are in the same sub-network response the discover packet by returning DHCP offer packet. In this state, the DHCP server offers the dynamic IP address for client (FPGA IP address). After receiving the offer packet, the client sends DHCP request packet to the server. Finally, the DHCP server returns DHCP ACK packet if the offered IP address is allocated for client device. Otherwise, the server returns DHCP NAK packet to reject the request from client device.

For releasing the IP address, the client device simply sends DHCP release packet with some identification. After that, the DHCP server de-allocates the IP address of the client device without sending any packets. More details about DHCP protocol, please refers to the following website.
https://tools.ietf.org/html/rfc2131

To receive DHCP packet, the filtering logic is configured by setting the value as shown in Figure 4-4.

♦ : Enable verification
♦ : Disable verification

Bits 0-7	Bits 8-15	Bits 16-23	Bits 24-31	Bits 32-39	Bits 40-47	Bits 48-55	Bits 56-63
Destination MAC Address						Source MAC Address	
Source MAC Address				Ethernet Type=IPv4		Version/HL = v4	Type of service
Length		Id Number		Fragment Offset		TTL	Protocol = UDP
IP Checksum		Source IP address				Destination IP address	
Destination IP address		Source port = DHCP server		Destination port = DHCP client		...	

Figure 4-4 DHCP packet filtering

Request IP address by DHCP

The sequence to run DHCP to get IP address from DHCP server (coded in dhcp_init_ipaddr function) is described as follows.

1) Display the latest (default) DHCP parameters on the console.
2) Receive user command to use the latest DHCP parameters or to update them before starting DHCP operating. Skip the step 3 if not updating the latest DHCP parameters.
3) Receive input parameters from user and check if input is valid or not. When the input is invalid, the parameter is not updated.
4) Set the filtering parameters of UserRxMAC to receive only DHCP packet by using init_filter function, as described in topic 4.2. The filtering data of DHCP packet contains the following fields (blue color in Figure 4-4).
   - Ethernet Type (2 bytes) = 0x0800 (IPv4)
   - IP version (1 byte) = 0x45 (Version 4)
   - Protocol (1 byte) = 0x11 (UDP Protocol)
   - Destination IP Address (4 bytes) = IP address of FPGA
   - Source port (2 byte) = 0x0043 (DHCP server port)
   - Destination port (2 bytes) = 0x0044 (DHCP client port)

   Note: Figure 4-4 shows only 38-byte of DHCP packet, the actual size of DHCP packet is larger than 38 bytes because the filtering logic in UserRxMAC module is designed to support up to 38-byte header data.
5) Enable receive data mode of UserRxMac module (RXEMAC_CMD_INTREG[0]='1').
6) Start the timer for packet recovery of the DHCP discover packet.
7) Prepare DHCP discover packet in the transmit temporal buffer (txbuff_ch) which is a global variable in CPU. Calculate length of the packet and set MAC/IP addresses of the DHCP discover packet. After that, use "write_udpip" function to create UDP/IP header, copy data from the transmit temporal buffer (txbuff_ch) to TxRAM (TXRAM_BASE_AD), and start data transmission of TxEMAC by setting TXMAC_LEN_INTREG=packet length.
8) Wait until the DHCP offer packet is stored in RxRAM by using "prepare_rxbuffer" function (checking the empty flag of RxMacFifo). If the timeout is found before receiving the offer packet, resend the DHCP discover packet by stopping the timer and then returning to step 6. Otherwise, continue to the next step for decoding the received packet.
9) The "prepare_rxbuffer" reads the last address of received packet from RxMacFifo and then copies the received data from RxRAM to receive temporal buffer (rxbuff_ch) in the firmware. After that, the received packet is decoded. Continue the next step if the received packet is the DHCP offer packet and the parameters including checksum are correct. Otherwise, return to step 8) to wait for the next received packet.
10) Reset the timer for packet recovery of the DHCP request packet. Next, initialize the retransmission counter for the DHCP request process.
11) Prepare DHCP request packet in the transmit temporal buffer, similar to the step 7.
12) Wait until the DHCP request packet is stored in RxRAM by using "prepare_rxbuffer" function.
    a. If timeout is found and total retry times (Retransmission counter=4), terminate DHCP operation and returning to step 1).
    b. If the timeout is found, restart the timer and return to step 11). Also, increase the retransmission counter.
    c. Continue the next step if some packet is received.

13) Similar to step 9), the received packet is decoded.
    a. If the received packet is DHCP ACK packet, continue the next step.
    b. If the received packet is DHCP NAK packet, terminate the DHCP operation by returning to step 1).
    c. Otherwise, return to step 12) to wait for the next received packet.
14) Display the DHCP completion message and the DHCP parameters that obtains from DHCP server on the console. Now, the IP address is obtained successfully.

Release IP address by DHCP

The sequence to run DHCP to release IP address from DHCP server (coded in dhcp_release_ipaddr function) is described as follows.

1) DHCP release packet is prepared by using the same procedure as step 7) of Request IP address operation.
2) Display the completion message to acknowledge the user that IP address is released. Next, wait for 2 seconds to delay the re-initialization process of DHCP operation.
3) Call “dhcp_init_ipaddr” function to get the IP address by DHCP.
4) Start TOE100G-IP initialization process, similar to menu [1] Reset TOE100G-IP. The operation is completed after the TOE100G-IP finishes the initialization process.

4.2 Function list in User application

This topic describes the function list to run the DHCP firmware. Similar to the Ping firmware, the function list is split into two groups – fast-port connection and slow-port connection. The details of fast-port connection function are similar to topic 3.2.1. The details of the function for operating slow-port connection by CPU in DHCP application are described as follows.

There are the basic functions for operating slow-port connection which are similar to Ping demo, i.e., cal_checksum for checksum calculating and prepare_rxbuffer to scan the received packet. Please see the details of these functions in topic 3.2.2.

int dhcp_init_ipaddr (void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	The function to request IP address by DHCP, as described in topic 4.1.

void dhcp_release_ipaddr (void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	The function to release IP address by DHCP and then re-initialize the system, as described in topic 4.1.

void init_filter (void)	
Parameters	None
Return value	None
Description	Write RXEMAC_CMD_INTREG to pause the storing of received packet in the slow-port connection. Next, Write RXEMAC_HDVAL_ADDR and RXEMAC_HDEN_ADDR for filtering a DHCP packet from DHCP server as shown in blue color text of Figure 4-4.

unsigned int verify_rx_dhcp (unsigned int *payload_len)	
Parameters	payload_len: Unsinged int pointer to return the DHCP packet length.
Return value	1: The received packet is DHCP based UDP/IP packet and the packet is valid. 0: The received data packet is not DHCP packet or the packet is not valid.
Description	Calculate the received DHCP packet length in byte-unit, verify the 16-bit checksum of IP and UDP packet, and verify the general field of DHCP reply packet. *Note: The packet is processed by using character array which is prepared by prepare_rxbuffer function.*

void write_udpip (unsigned int payload_len)	
Parameters	payload_len: the length of the payload data in byte-unit
Return value	None
Description	Create the UDP/IP packet by using the transmitted payload data (the global variable) and its length (the input parameter). The parameters such as UDP port numbers, IP addresses, and Ethernet MAC addresses are set to prepare the Ethernet packet for DHCP operation. After that, wait until UserTxMAC is Idle (TXEMAC_LEN_INTREG[0]='0') and then copy the prepared Ethernet packet to TxRAM (TXRAM_BASE_ADDR). Finally, Start transmitting the packet by setting TXEMAC_LEN_INTREG = packet length value.

4.3 Test software on PC (DHCP server application)

"dhcpsrv" application is the software which implements a DHCP Server for Windows based system. It is used to assign IP addresses to client device. The test environment uses this software to check DHCP operation of slow-port connection. The application can be downloaded from the website.
https://www.dhcpserver.de/cms/

The test environment uses 2.5.2 version. There are four objects in the downloaded file.

- dhcpsrv.exe - the DHCP server program.
- dhcpwiz.exe - the DHCP server configuration program.
- wwwroot - the folder of basic web files.
- readme.txt - information about the other files in an archive.

It is recommended to run "dhcpwiz.exe" for configuring the server and writing the script file with the graphical user interface. More details of configuring the DHCP server or running the server in advance mode are provided in the contributor website as the above link.

5 Revision History

Revision	Date	Description
1.0	26-May-22	Initial version release