TOE100G-IP with CPU reference design

Rev1.2 25-May-22

1 Introduction

TCP/IP is the core protocol of the Internet Protocol Suite for networking application. TCP/IP model has four layers, i.e., Application Layer, Transport Layer, Internet Layer, and Network Access Layer. As shown in Figure 1-1, five layers are displayed for simply matching with the hardware implementation on FPGA. Network Access Layer is split into Link and Physical Layer.

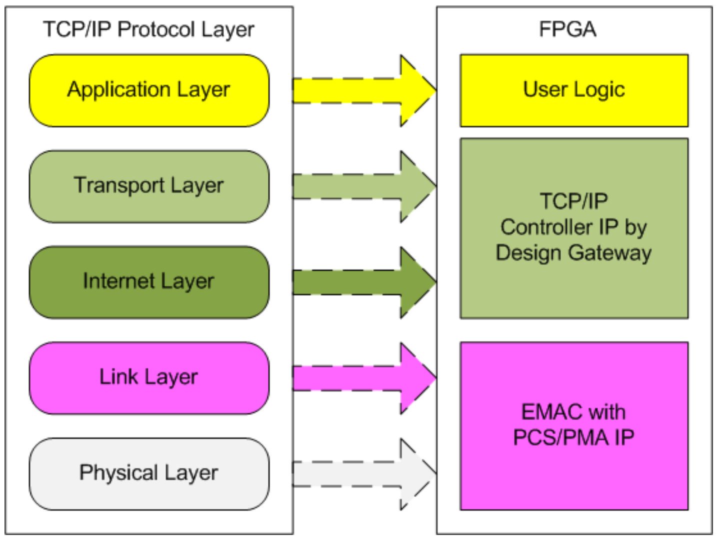


Figure 1-1 TCP/IP protocol layer

TOE100G-IP implements the Transport and Internet layer of TCP/IP Protocol for building Ethernet packet from the user data which is TCP payload data to EMAC. If TCP payload data size is larger than a packet size, TOE100G-IP splits the data to send by using multiple packets. Next, the TCP payload data is appended by TCP/IP header. On the other hand, the received Ethernet packet from EMAC is extracted by TOE100G-IP. The header of the packet is verified. If the header is valid, TCP payload data is forwarded to the user logic. Otherwise, the packet is rejected.

The lower layer protocols are implemented by 100G Ethernet IP, including EMAC, PCS, and PMA logic. The reference design uses 100G Ethernet (MAC) Subsystem, provided by Xilinx.

The reference design provides the evaluation system which includes simple user logic to transfer data by using TOE100G-IP. TOE100G-IP supports to transfer data with PC or another TOE100G-IP run on another FPGA board. To run with PC, the test application is called on PC to send and verify TCP payload data via Ethernet connection at very high-speed rate. Two test applications are specially designed, “tcpdatatest” for running half-duplex test (send or receive data test) and “tcp_client_txrx_40G” for running full-duplex test (send and receive data at the same time by one session).

To allow the user setting the test parameters and controlling the operation of TOE100G-IP demo via UART, the CPU system is included. It is easy for the user to set the test parameters and monitor the current status on the console. The firmware on CPU is built by using bare-metal OS. More details of the demo are described as follows.

2 Hardware overview

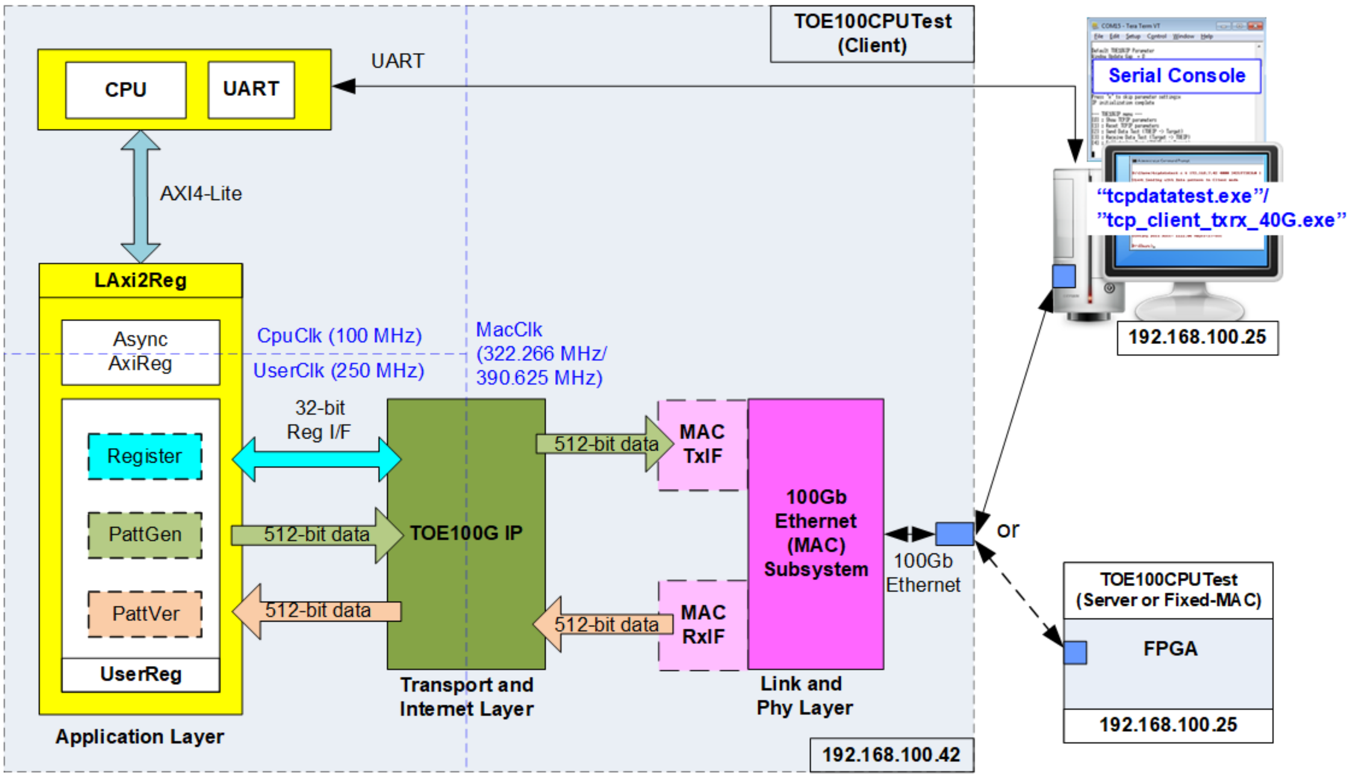


Figure 2-1 Demo block diagram

In test environment, two devices are used for 100Gb Ethernet transferring. When running by FPGA and PC, FPGA is initialized by Client mode and PC is initialized by Server mode. On the other hand, when running by two FPGAs, it may be initialized by Client <–> Server, Client <–> Fixed-MAC, or Fixed-MAC <-> Fixed-MAC, as shown in Figure 2-1. When using PC, the test applications (tcpdatatest and tcp_client_txrx_40G) must be run on PC for transferring data.

In FPGA system, TOE100G-IP connects with 100G Ethernet (MAC) Subsystem to implement all TCP/IP layers. For UltraScale+ device, 100G Ethernet Subsystem can connect with TOE100G-IP directly through 512-bit AXI4-Stream interface. For Versal device, 100G Ethernet MAC Subsystem uses 384-bit AXI4-Stream interface, so it needs to design MACTxIF and MacRxIF to convert data width for connecting with TOE100G-IP. User interface of TOE100G-IP connects to UserReg within LAxi2Reg. UserReg consists of Register file for interfacing with Register interface, PattGen for sending test data via Tx FIFO interface, and PattVer for verifying test data via Rx FIFO interface. Register files of UserReg are controlled by CPU firmware through AXI4-Lite bus.

There are three clock domains in the design, i.e., CpuClk which is the clock for running the CPU system, MacClk which is the clock for user interface 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.

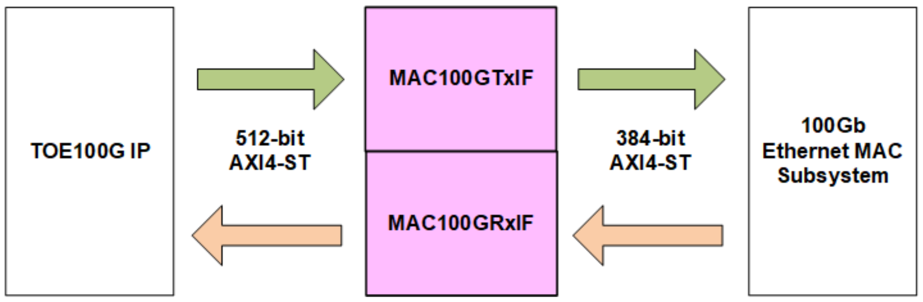


Figure 2-2 Adapter logic of EMAC interface in Versal Device

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.

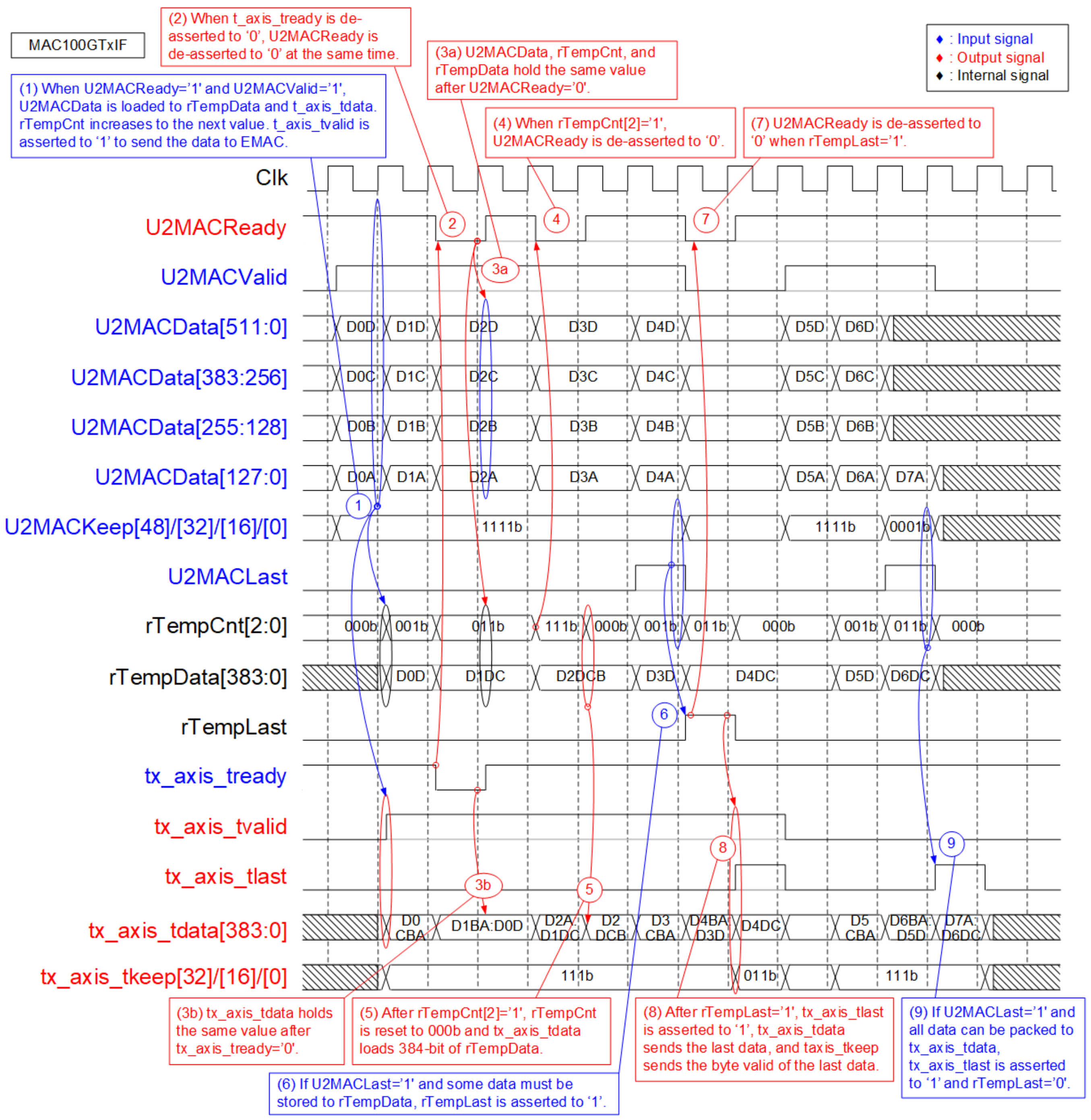

MAC100GTxIF
♦ : Input signal
♦ : Output signal
♦ : Internal signal
(1) When U2MACReady='1' and U2MACValid='1', U2MACData is loaded to rTempData and t_axis_tdata. rTempCnt increases to the next value. t_axis_tvalid is asserted to '1' to send the data to EMAC.
(2) When t_axis_tready is de-asserted to '0', U2MACReady is de-asserted to '0' at the same time.
(3a) U2MACData, rTempCnt, and rTempData hold the same value after U2MACReady='0'.
(3b) tx_axis_tdata holds the same value after tx_axis_tready='0'.
(4) When rTempCnt[2]='1', U2MACReady is de-asserted to '0'.
(5) After rTempCnt[2]='1', rTempCnt is reset to 000b and tx_axis_tdata loads 384-bit of rTempData.
(6) If U2MACLast='1' and some data must be stored to rTempData, rTempLast is asserted to '1'.
(7) U2MACReady is de-asserted to '0' when rTempLast='1'.
(8) After rTempLast='1', tx_axis_tlast is asserted to '1', tx_axis_tdata sends the last data, and tax is_tkeep sends the byte valid of the last data.
(9) If U2MACLast='1' and all data can be packed to tx_axis_tdata, tx_axis_tlast is asserted to '1' and rTempLast='0'.
Clk
U2MACReady
U2MACValid
U2MACData[511:0]
U2MACData[383:256]
U2MACData[255:128]
U2MACData[127:0]
U2MACKeep[48]/[32]/[16]/[0]
U2MACLast
rTempCnt[2:0]
rTempData[383:0]
rTempLast
tx_axis_tready
tx_axis_tvalid
tx_axis_tlast
tx_axis_tdata[383:0]
tx_axis_tkeep[32]/[16]/[0]
D0D D1D D2D D3D D4D D5D D6D
D0C D1C D2C D3C D4C D5C D6C
D0B D1B D2B D3B D4B D5B D6B
D0A D1A D2A D3A D4A D5A D6A D7A
1111b 1111b 0001b
000b 001b 011b 111b 000b 001b 011b 000b 001b 011b 000b
D0D D1DC D2DCB D3D D4DC D5D D6DC
D0 CBA D1BA:D0D D2A D1DC D2 DCB D3 CBA D4BA D3D D4DC D5 CBA D6BA D5D D7A D6DC
111b 011b 111b

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.
   <u>Note</u>*: 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).

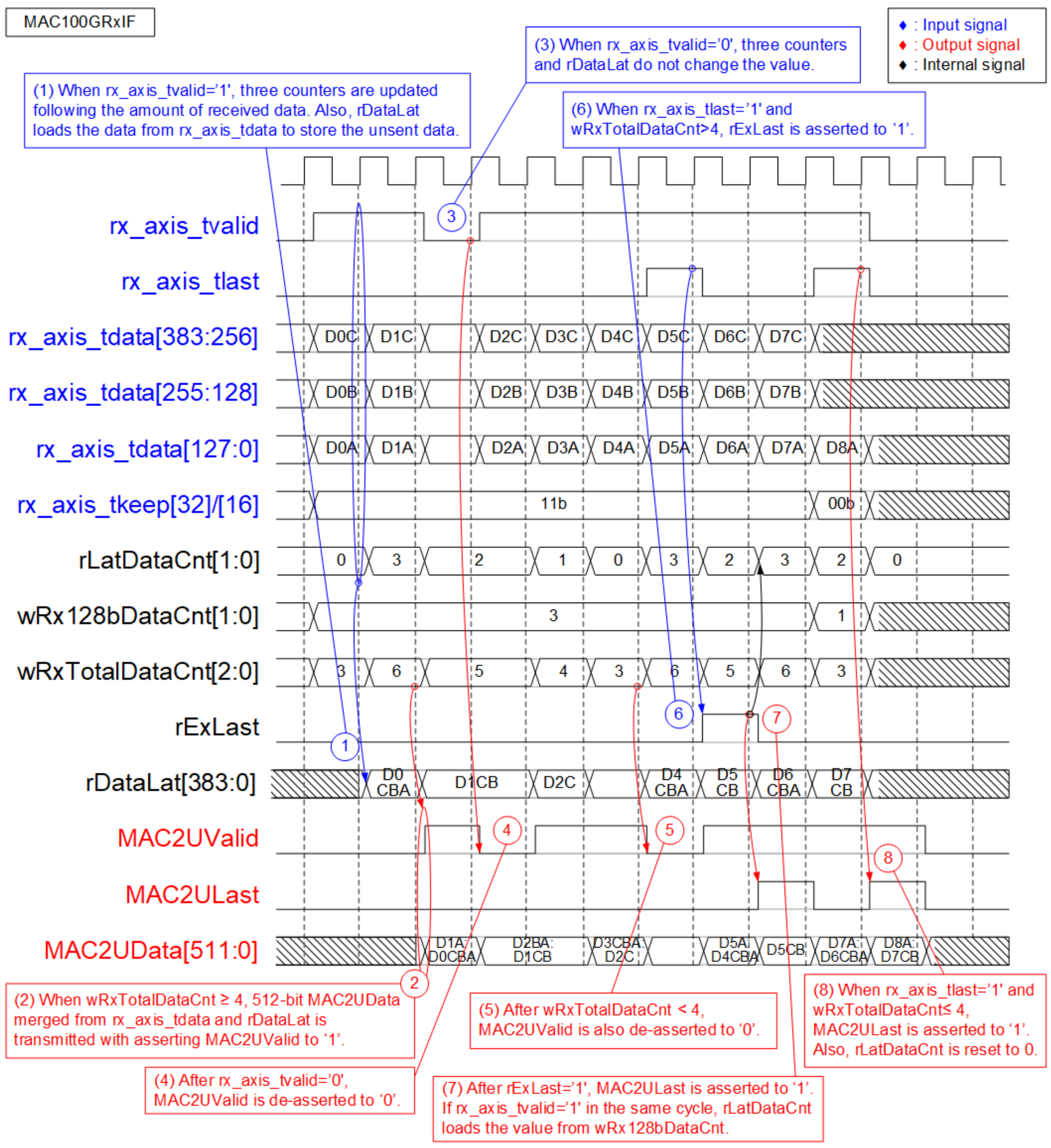
384-bit latch register (rDataLat) uses rLatDataCnt to determine the maximum number of received data from EMAC that must be kept in the next cycle.

(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.



(1) When rx_axis_tvalid='1', three counters are updated following the amount of received data. Also, rDataLat loads the data from rx_axis_tdata to store the unsent data.

(2) When wRxTotalDataCnt ≥ 4, 512-bit MAC2UData merged from rx_axis_tdata and rDataLat is transmitted with asserting MAC2UValid to '1'.

(3) When rx_axis_tvalid='0', three counters and rDataLat do not change the value.

(4) After rx_axis_tvalid='0', MAC2UValid is de-asserted to '0'.

(5) After wRxTotalDataCnt < 4, MAC2UValid is also de-asserted to '0'.

(6) When rx_axis_tlast='1' and wRxTotalDataCnt>4, rExLast is asserted to '1'.

(7) After rExLast='1', MAC2ULast is asserted to '1'. If rx_axis_tvalid='1' in the same cycle, rLatDataCnt loads the value from wRx128bDataCnt.

(8) When rx_axis_tlast='1' and wRxTotalDataCnt≤ 4, MAC2ULast is asserted to '1'. Also, rLatDataCnt is reset to 0.

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 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

2.3 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 2-5.

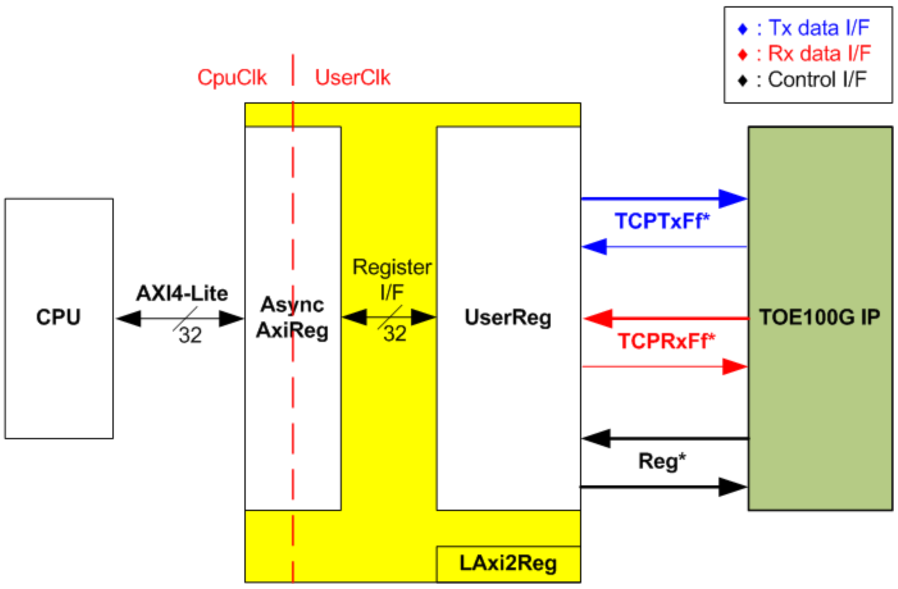


Figure 2-5 LAxi2Reg block diagram

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 includes the register file of the parameters and the status signals of test logics. Both data interface and control interface of TOE100G-IP are also connected to UserReg. More details of AsyncAxiReg and UserReg are described as follows.

2.3.1 AsyncAxiReg

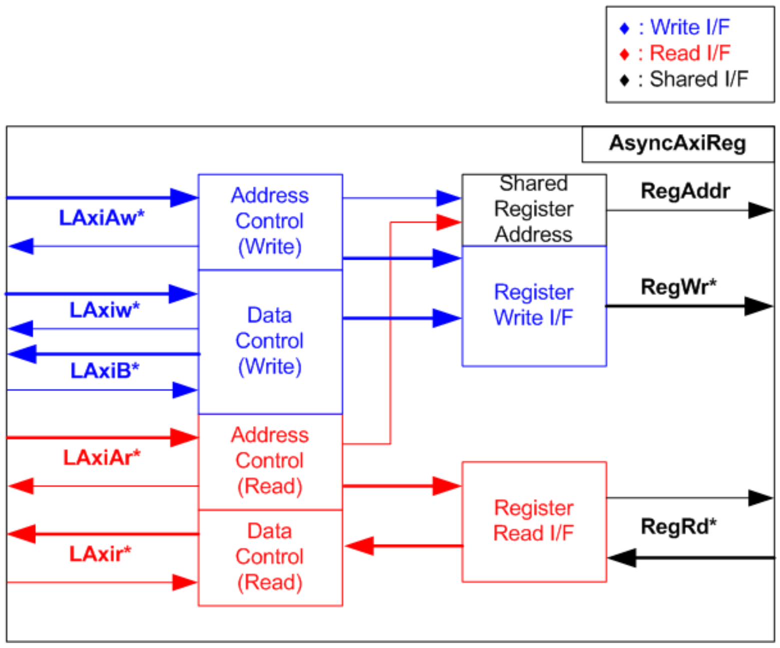


Figure 2-6 AsyncAxiReg interface

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 2-6. 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 2-7.

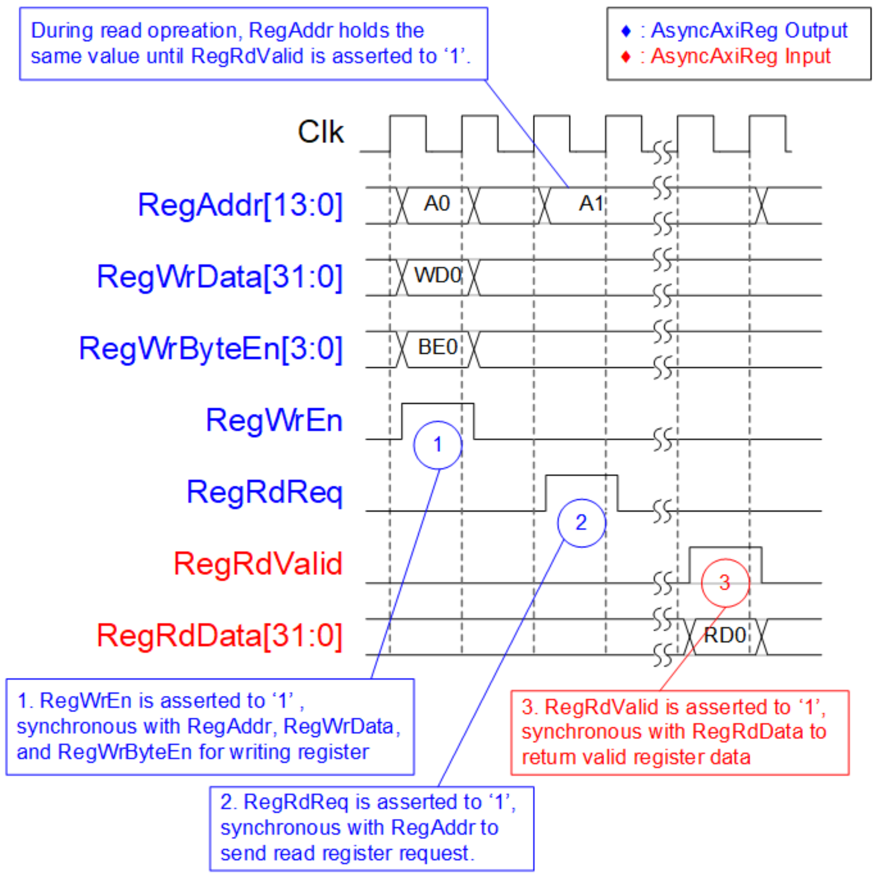

During read opreation, RegAddr holds the same value until RegRdValid is asserted to '1'.
♦ : AsyncAxiReg Output
♦ : AsyncAxiReg Input
Clk
RegAddr[13:0]
A0
A1
RegWrData[31:0]
WD0
RegWrByteEn[3:0]
BE0
RegWrEn
RegRdReq
RegRdValid
RegRdData[31:0]
RD0
1
2
3
1. RegWrEn is asserted to '1' , synchronous with RegAddr, RegWrData, and RegWrByteEn for writing register
2. RegRdReq is asserted to '1', synchronous with RegAddr to send read register request.
3. RegRdValid is asserted to '1', synchronous with RegRdData to return valid register data

Figure 2-7 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.

2.3.2 UserReg

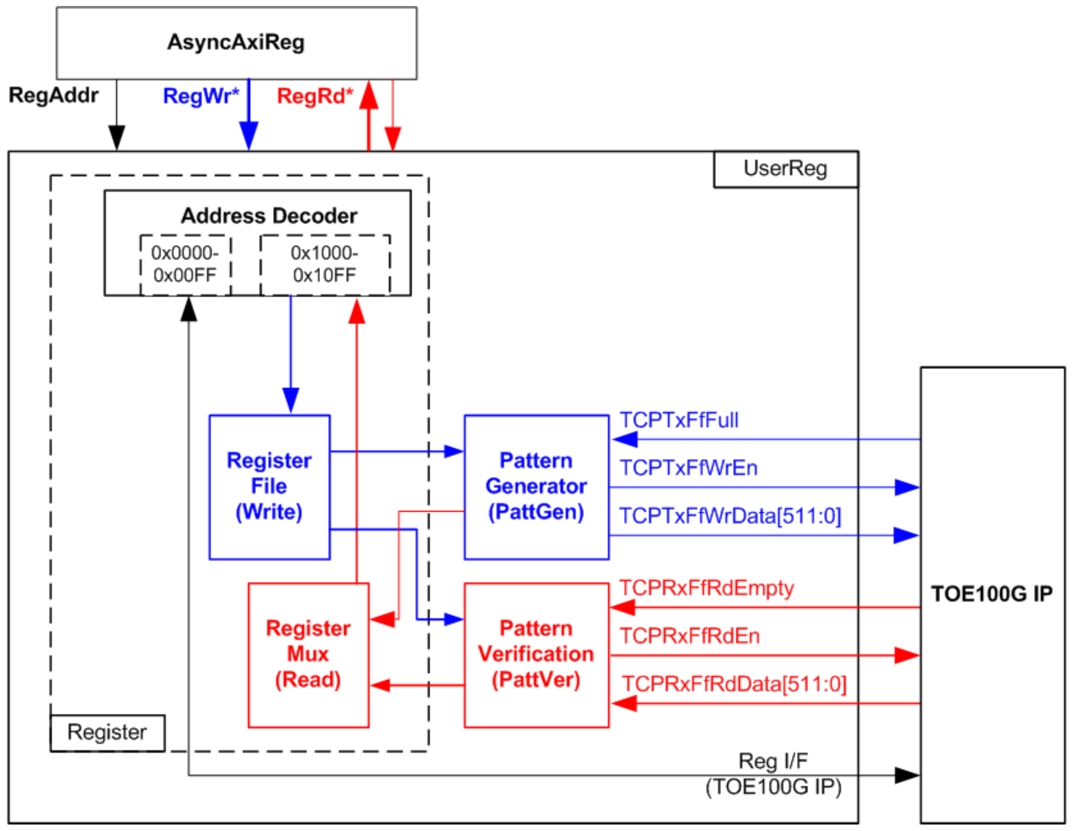


Figure 2-8 UserReg block diagram

The logic inside UserReg has three operations, i.e., Register, Pattern generator (PattGen), and Pattern verification (PattVer). Register block decodes the address requested from AsyncAxiReg and then selects the active register for write or read transaction. Pattern generator block is designed to send 512-bit test data to TOE100G-IP following FIFO interface standard. Pattern verification block is designed to read and verify 512-bit data from TOE100G-IP following FIFO interface standard. More details of each block are described as follows.

Register Block

The address range, mapped to UserReg, is split into two areas, i.e., TOE100G-IP register (0x0000-0x00FF) and UserReg register (0x1000-0x10FF). Therefore, the upper bit of RegAddr is applied to select the active area between TOE100G-IP or UserReg register. While the lower bits of RegAddr are fed to TOE100G-IP and internal registers of UserReg to select the active register. The registers inside UserReg are 32-bit data, so write byte enable (RegWrByteEn) is not used. To write hardware registers, the CPU must use 32-bit pointer to place 32-bit valid value on the write data bus.

To read register, multiplexer selects the read data within each address area. The lower bit of RegAddr is applied in each Register area to select the data. Next, the address decoder uses the upper bit to select the read data from each area for returning to CPU. Totally, the latency of read data is equal to one clock cycle, so RegRdValid is created by RegRdReq with asserting one D Flip-flop. More details of the address mapping within UserReg module are shown in Table 2-1.

Table 2-1 Register map Definition

Address Wr/Rd	Register Name (Label in the "toe100gtest.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'. [16] – Ethernet linkup status from Ethernet MAC ('0': Not linkup, '1': Linkup)
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 from TOE100G-IP in 512-bit unit. The value is cleared to 0 when USER_CMD_INTREG is written by user.
BA+0x1080 Rd	EMAC IP version (EMAC_VER_INTREG)	Rd[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.

Pattern Generator

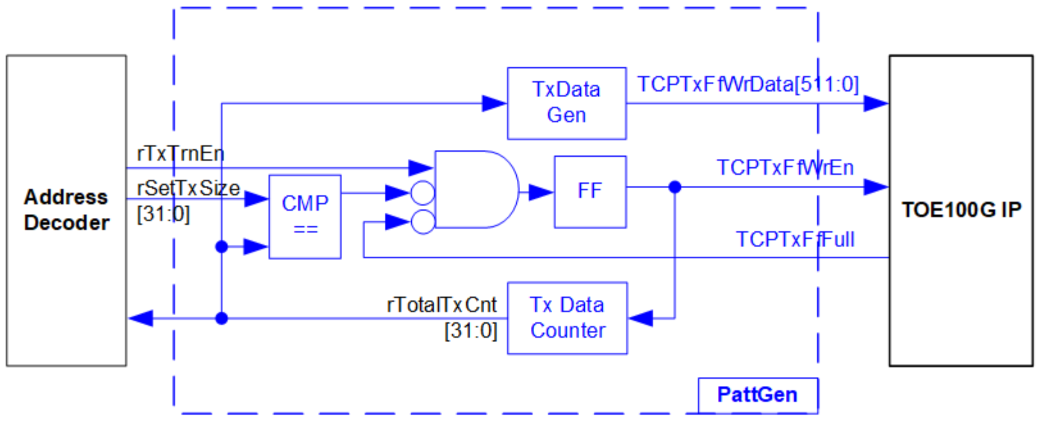


Figure 2-9 PattGen block

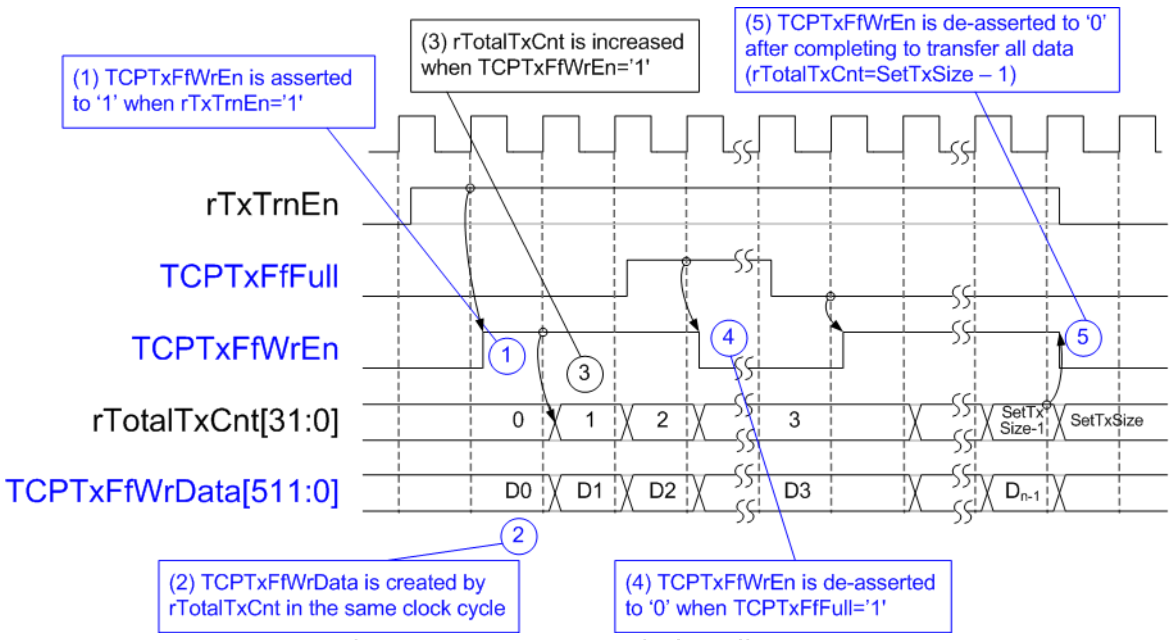

(1) TCPTxFfWrEn is asserted to '1' when rTxTrnEn='1'
(3) rTotalTxCnt is increased when TCPTxFfWrEn='1'
(5) TCPTxFfWrEn is de-asserted to '0' after completing to transfer all data (rTotalTxCnt=SetTxSize – 1)
rTxTrnEn
TCPTxFfFull
TCPTxFfWrEn
rTotalTxCnt[31:0]
TCPTxFfWrData[511:0]
0
1
2
3
SetTxSize-1
SetTxSize
D0
D1
D2
D3
Dn-1
(2) TCPTxFfWrData is created by rTotalTxCnt in the same clock cycle
(4) TCPTxFfWrEn is de-asserted to '0' when TCPTxFfFull='1'

Figure 2-10 PattGen timing diagram

Figure 2-9 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 2-10.

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

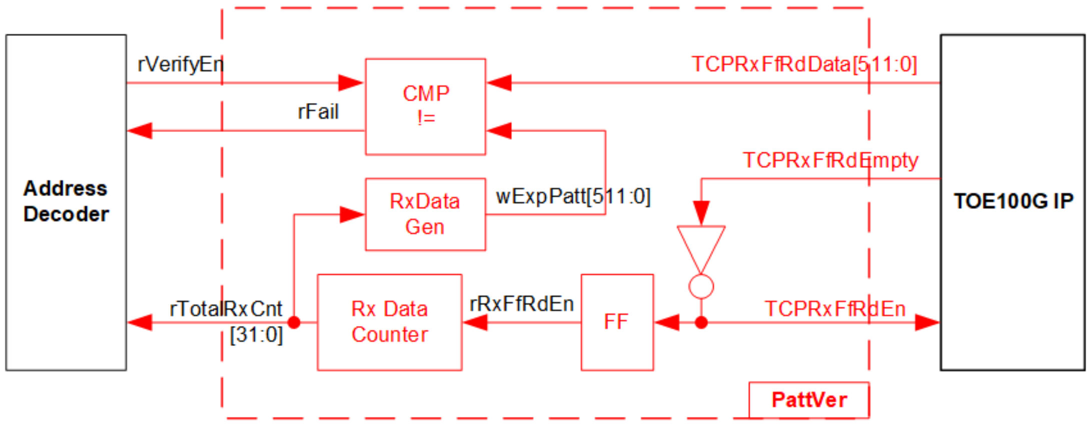
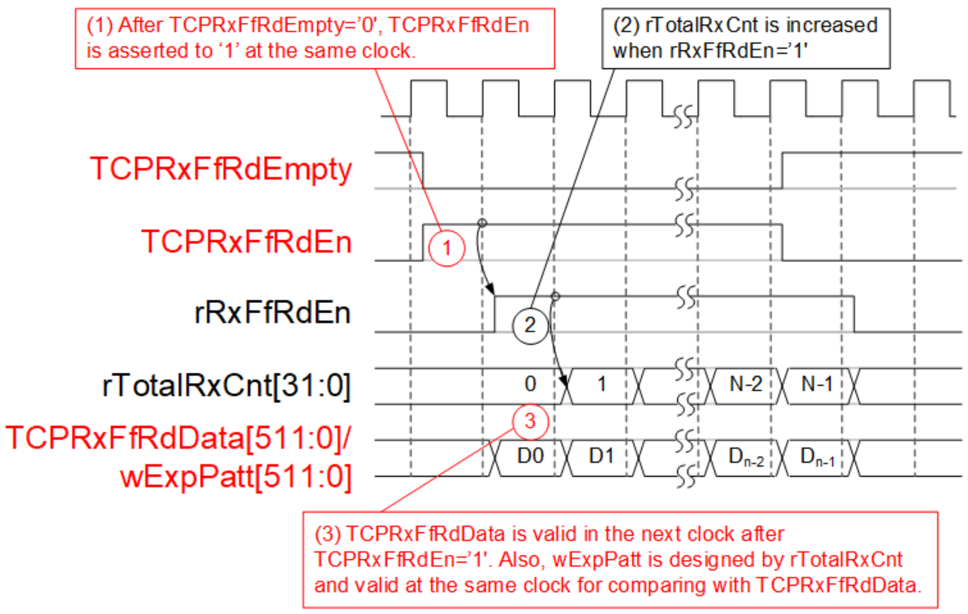


Figure 2-11 PattVer block



(1) After TCPRxFfRdEmpty='0', TCPRxFfRdEn is asserted to '1' at the same clock.
(2) rTotalRxCnt is increased when rRxFfRdEn='1'
TCPRxFfRdEmpty
TCPRxFfRdEn
rRxFfRdEn
rTotalRxCnt[31:0]
TCPRxFfRdData[511:0]/ wExpPatt[511:0]
1
2
3
0
1
N-2
N-1
D0
D1
Dn-2
Dn-1
(3) TCPRxFfRdData is valid in the next clock after TCPRxFfRdEn='1'. Also, wExpPatt is designed by rTotalRxCnt and valid at the same clock for comparing with TCPRxFfRdData.

Figure 2-12 PattVer Timing diagram

Figure 2-11 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 2-12.

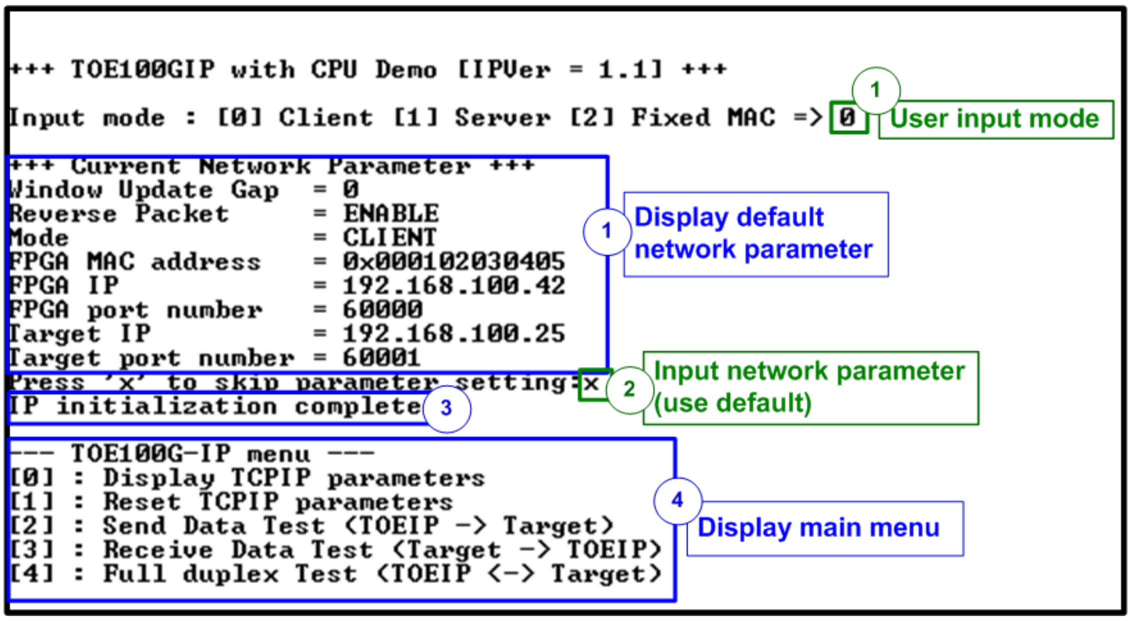
When rVerifyEn is set to '1', data comparison is enabled to compare read data (TCPRxFfRdData) with the expected pattern (wExpPatt). If data verification is failed, rFail is asserted to '1'. TCPRxFfRdEn is designed by using NOT logic of TCPRxFfRdEmpty. TCPRxFfRdData is valid for data comparison in the 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 on FPGA

After FPGA boot-up, 100G Ethernet link up status (USER_RST_INTREG[16]) 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.

- When testing by FPGA and PC, it is recommended to initialize TOE100G-IP in Client mode. After PC receives ARP request from TOE100G-IP, PC returns ARP reply. It is not simple to force PC sending ARP request to complete FPGA initialization in Server mode.
- When testing by two FPGAs, the initialization mode on two FPGAs must be following rules. If the first board sets Server mode, another board must be set to Client mode. Otherwise, the first board is set to Fixed-MAC mode while another board is set to Client or Fixed-MAC mode.

After receiving the mode from the user, the default parameters in the selected mode are displayed on the console. The user can select to complete the initialization by using default parameters or updated parameters. The example when the system is initialized in Client mode by using default parameters is shown in Figure 3-1.



```
+++ TOE100GIP with CPU Demo [IPVer = 1.1] +++
Input mode : [0] Client [1] Server [2] Fixed MAC => 0
+++ Current Network Parameter +++
Window Update Gap   = 0
Reverse Packet      = ENABLE
Mode                = CLIENT
FPGA MAC address    = 0x000102030405
FPGA IP             = 192.168.100.42
FPGA port number    = 60000
Target IP           = 192.168.100.25
Target port number  = 60001
Press 'x' to skip parameter setting:x
IP initialization complete
--- TOE100G-IP menu ---
[0] : Display TCPIP parameters
[1] : Reset TCPIP parameters
[2] : Send Data Test (TOEIP -> Target)
[3] : Receive Data Test (Target -> TOEIP)
[4] : Full duplex Test (TOEIP <-> Target)
```

1
User input mode
1
Display default network parameter
2
Input network parameter (use default)
3
4
Display main menu

Figure 3-1 System initialization in Client mode by using default parameters

There are four steps to complete initialization process 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 3.2).
3) CPU waits until TOE100G-IP finishes initialization process (TOE_CMD_INTREG[0]='0').
4) Main menu is displayed. There are five test operations for user selection. More details of each menu are described as follows.

3.1 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.

3.2 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').

3.3 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/FPGA. Data is verified by Test application on PC (in case of PC <-> FPGA) or verification module in FPGA (in case of FPGA <-> FPGA). The operation is finished when total data are transferred from FPGA to PC/FPGA. 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 or FPGA). 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.

3.4 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 or FPGA) 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 or FPGA). 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 receive length of user logic (USER_RXLEN_INTREG) with the set value from user. 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.

3.5 Full duplex test

This menu is designed to run full duplex test by transferring data between FPGA and another device (PC/FPGA) 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).

When running the test by using PC, the transfer size set on FPGA must be matched to the size set on test application (tcp_client_txrx_40G). Connection mode on FPGA when running with PC 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.

3.6 Function list in User application

This topic describes the function list to run TOE100G-IP operation.

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 3.4

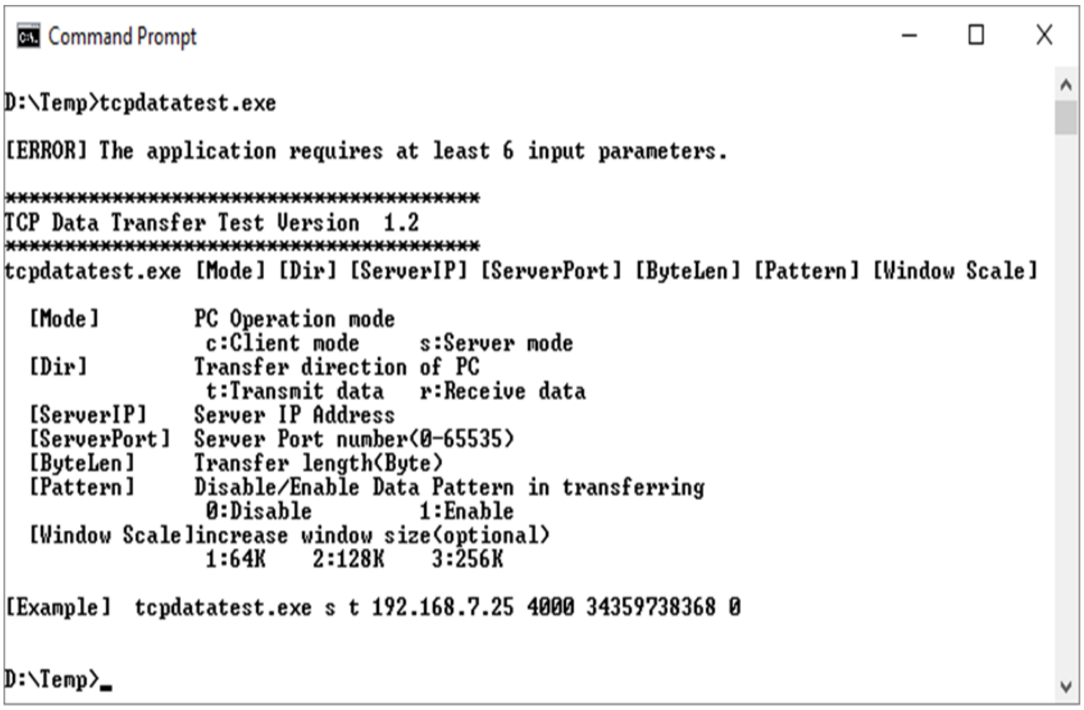
int toe_send_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Send data test following description in topic 3.3

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 3.5

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

4 Test Software on PC

4.1 “tcpdatatest” for half duplex test


Command Prompt

```
D:\Temp>tcpdatatest.exe

[ERROR] The application requires at least 6 input parameters.

****************************************
TCP Data Transfer Test Version  1.2
****************************************
tcpdatatest.exe [Mode] [Dir] [ServerIP] [ServerPort] [ByteLen] [Pattern] [Window Scale]

  [Mode]        PC Operation mode
                 c:Client mode     s:Server mode
  [Dir]         Transfer direction of PC
                 t:Transmit data   r:Receive data
  [ServerIP]    Server IP Address
  [ServerPort]  Server Port number(0-65535)
  [ByteLen]     Transfer length(Byte)
  [Pattern]     Disable/Enable Data Pattern in transferring
                 0:Disable         1:Enable
  [Window Scale]increase window size(optional)
                 1:64K    2:128K    3:256K

[Example]  tcpdatatest.exe s t 192.168.7.25 4000 34359738368 0


D:\Temp>_
```

Figure 4-1 “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: Optional parameter which is not used in the demo.

Transmit data mode

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

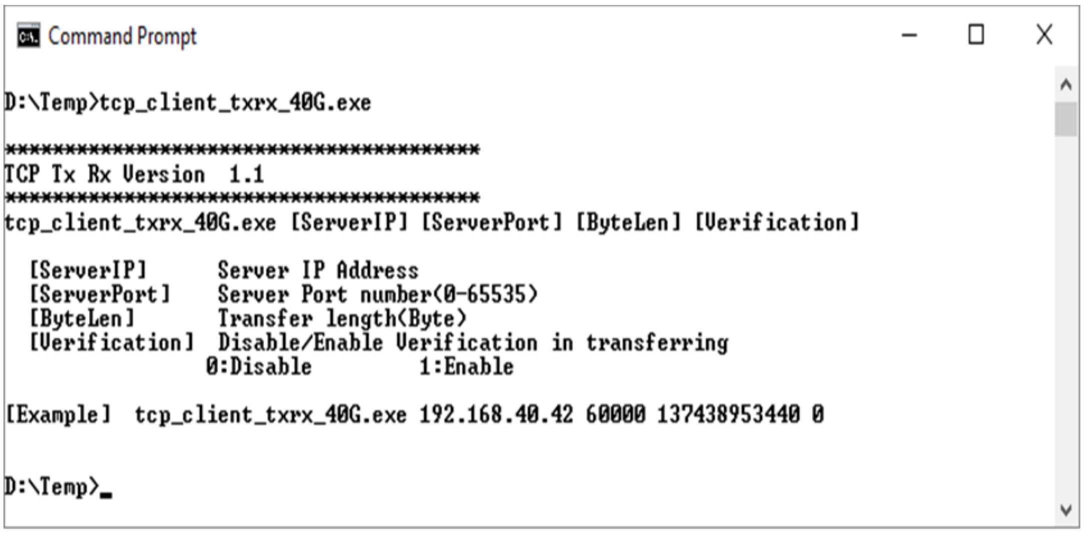
1) Get parameters from the user and verify that all inputs are valid.
2) Create the socket and set socket options.
3) Create the new connection by using server IP address and server port number.
4) Allocate 1 MB memory to be send buffer.
5) Skip this step if the dummy pattern is selected. Otherwise, generate the incremental test pattern to send buffer.
6) Send data out and read total amount of 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.

4.2 “tcp_client_txrx_40G” for full duplex test



```
D:\Temp>tcp_client_txrx_40G.exe

*****************************************
TCP Tx Rx Version  1.1
*****************************************
tcp_client_txrx_40G.exe [ServerIP] [ServerPort] [ByteLen] [Verification]

  [ServerIP]      Server IP Address
  [ServerPort]    Server Port number(0-65535)
  [ByteLen]       Transfer length(Byte)
  [Verification]  Disable/Enable Verification in transferring
                 0:Disable        1:Enable

[Example]  tcp_client_txrx_40G.exe 192.168.40.42 60000 137438953440 0


D:\Temp>_
```

Figure 4-2 “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 4-2, 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.

5 Revision History

Revision	Date	Description
1.2	25-May-22	Support Ethernet MAC Subsystem
1.1	18-Mar-22	Add Reverse packet enable feature
1.0	25-Jan-21	Initial version release