



# TOE200GADV-IP reference design

Rev1.00 13-Jun-24

- 1 Introduction ..... 2
- 2 Hardware overview ..... 4
  - 2.1 F-Tile Ethernet Hard IP ..... 6
  - 2.2 QSFPCConfig..... 6
  - 2.3 TOEMACIF ..... 8
  - 2.4 AvlSSw2to1 ..... 8
  - 2.5 TOE200GADV-IP ..... 10
  - 2.6 User2MAC ..... 11
    - 2.6.1 UserTxMac..... 12
    - 2.6.2 UserRxMAC ..... 14
  - 2.7 CPU and Peripherals ..... 16
    - 2.7.1 AsyncAvlReg..... 17
    - 2.7.2 UserReg..... 19
- 3 CPU Firmware on FPGA ..... 32
  - 3.1 Display parameters..... 33
  - 3.2 Reset parameters ..... 33
  - 3.3 Half Duplex Test..... 34
  - 3.4 Full duplex test..... 36
  - 3.5 Ping reply test..... 38
  - 3.6 Function list in CPU firmware..... 40
    - 3.6.1 Function for High-Speed Connection ..... 40
    - 3.6.2 Functions for Low-Speed Connection ..... 45
- 4 Test Software on PC ..... 46
  - 4.1 'tcpdatatest' application (Half duplex test)..... 46
  - 4.2 'tcp\_client\_txrx\_single' application (Full duplex test)..... 48
- 5 Revision History ..... 50

# 1 Introduction

The TCP/IP (Transmission Control Protocol/Internet Protocol) suite is fundamental to network communications, facilitating data transfers across various networks with reliability. Typically, the host system utilizes a CPU to manage the TCP/IP stack via its operating system, which must also share resources among various tasks. Consequently, the performance of data transfer through a single TCP session is limited when relying on such CPU-based system.

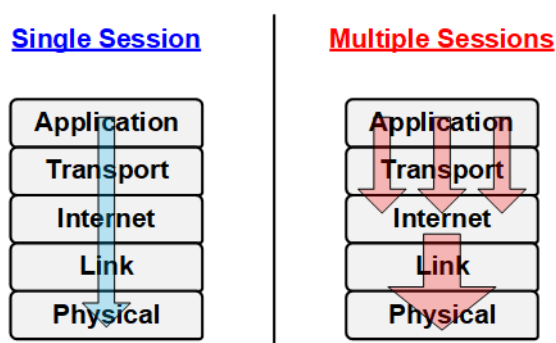


Figure 1-1 Multiple TCP session concept

To overcome this limitation, several researchers have proposed employing multiple TCP sessions. As shown in Figure 1-1, this concept creates a pipelined data flow in the upper layers, significantly increasing data volume and improving utilization of the lower layer bandwidth. However, this approach introduces another challenge: managing multiple TCP/IP sessions can consume significant CPU and OS resources.

To address these challenges, Design Gateway introduces the TOE200GADV-IP, a hardwired logic on an FPGA that implements the Transport and Internet layers of the TCP/IP Protocol. This IP Core fully offloads TCP/IP packet processing and incorporates a native multi-session architecture. It supports up to four simultaneous sessions on the same 200G Ethernet channel, significantly enhancing overall performance for data transfer between the FPGA and PC. Additionally, four TCP sessions can be utilized to transfer different data types simultaneously for the applications that require varied data types during system execution. In scenarios where data transfer occurs between two FPGAs via TOE200GADV-IP, using a single TCP session can achieve the peak performance of 200G Ethernet.

The TOE200GADV-IP is specifically designed for the rapid transmission of TCP payload data, making it well-suited for applications demanding ultra high-speed connectivity across all four TCP sessions. However, specific applications necessitate a designated port for the transfer of control information using alternative protocols like ICMP or DHCP, where high-speed transfer is not required. In response to this demand for lower-speed transfer, dedicated logic for CPU interface has been incorporated to optimize resource utilization and provide flexibility in handling varied processing requirements.

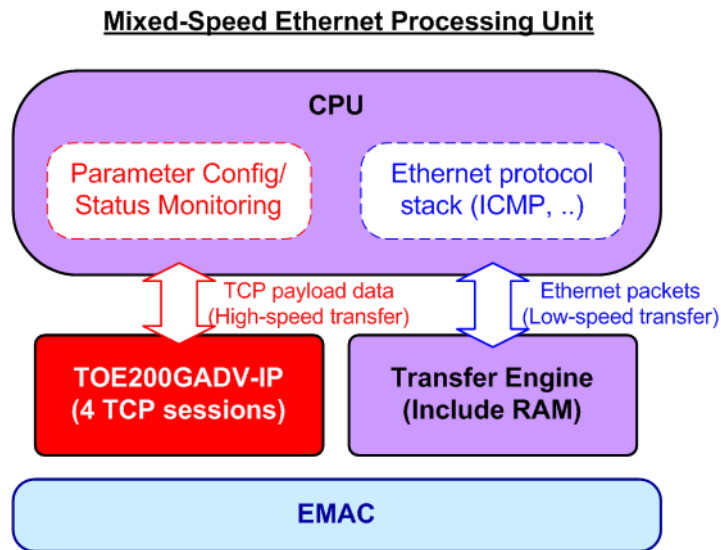


Figure 1-2 Mixed-speed and versatile Ethernet processing unit

The system illustrated in Figure 1-2 serves this purpose effectively. One TOE200GADV-IP module is deployed to manage four high-speed TCP ports, while the CPU takes charge of handling the remaining ports and other protocols that requires lower-speed processing.

This document outlines the reference design corresponding to the concept depicted in Figure 1-2. In this design, the CPU takes charge of handling the Ping command, using ICMP protocol, while the TOE200GADV-IP is integrated to process four high-speed TCP payload data. Although the reference design activates all four sessions of TOE200GADV-IP, users have the flexibility to enable each session independently. This feature facilitates performance evaluation and operational testing with fewer than four sessions. Additionally, the transfer direction of each session can be individual configured to meet specific requirements. User can also customize the multi-session reference design by adjusting the number of sessions as needed.

For enhanced demo flexibility, a UART interface is integrated with the CPU system to establish a user console. This console enables users to set test parameters, control demo operations, and monitor the current test status. The CPU firmware is developed using a simple bare-metal OS. Further details of the reference design are described in the subsequent sections.

## 2 Hardware overview

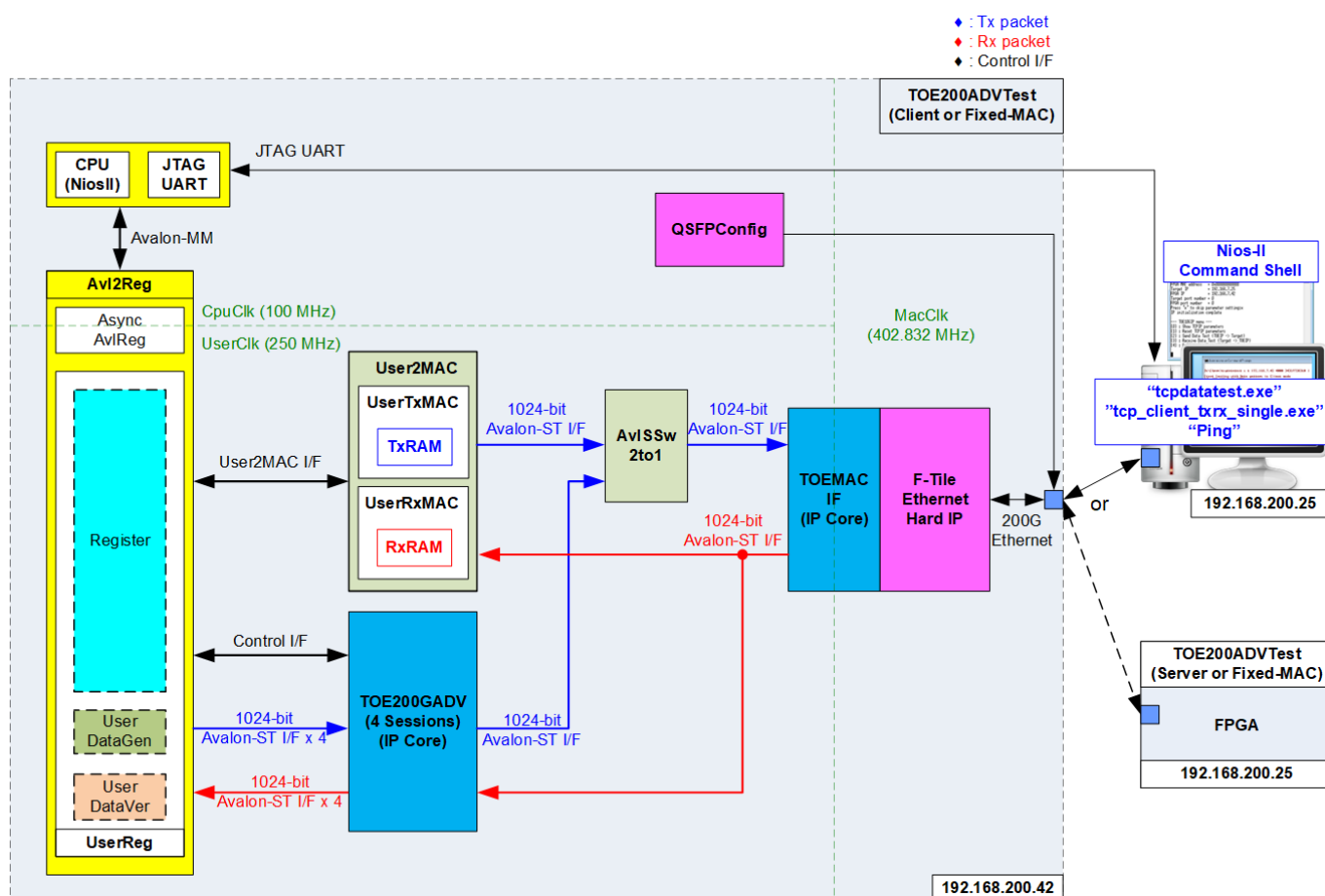


Figure 2-1 Demo block diagram

In the test environment, two devices are utilized for the transfer of 200G Ethernet data. When utilizing an FPGA and a PC, the FPGA is initialized in Client mode, while the PC is initialized in Server mode. Conversely, with two FPGAs, initialization can occur in one of the following modes: Client <=> Server, Client <=> Fixed-MAC, or Fixed-MAC <=> Fixed-MAC, as depicted in Figure 2-1. Three test applications run on the PC, facilitating the transfer of TCP payload data ('tcpdatatest.exe' and 'tcp\_client\_trrx\_single.exe') or the transfer of a Ping command.

The reference design offers two connection types: a low-speed connection managed by the CPU and a high-speed connection facilitated by TOE200GADV-IP. For the low-speed connection, the CPU firmware implements an ICMP protocol for Ping command testing. The Ethernet packet for this connection is transferred through User2MAC, with parameters configured by the CPU to exclusively handle ICMP packets. User2MAC comprises TxRAM and RxRAM to store Ethernet packets transferring to/from the Ethernet Hard IP. UserReg serves as the interface for CPU access to TxRAM and RxRAM.

For the high-speed connection, TOE200GADV-IP is integrated to process TCP payload data for four sessions. The CPU configures the parameters of TOE200GADV-IP through UserReg, the 32-bit Reg I/F. The data interface of the four TCP sessions is managed by UserDataGen or UserDataVer, depending on the transfer direction.

The system has the capability to concurrently process Ethernet packets from high-speed and low-speed connections. However, the main menu in the CPU firmware allows users to choose between executing high-speed or low-speed connection for basic operational testing. The AvlSSw2to1 module functions as the switch logic, selecting the source of transmitted packets to the TOEMACIF, which can be User2MAC or TOE200GADV-IP. The receive interface of the TOEMACIF is directly connected to both User2MAC and TOE200GADV-IP, each incorporating their packet filtering logics to selectively bypass specified packets.

To connect to the 200G Ethernet, two modules are utilized in the lower layer: QSFPConfig to configure the QSFP transceiver and the F-Tile Ethernet Hard IP serving as the Ethernet MAC. The F-Tile Ethernet Hard IP is an integrated hard IP in Intel FPGAs, featuring a 512-bit MAC Segmented user interface. However, this interface does not match the 1024-bit Avalon stream interface used by the TOE200GADV-IP and User2MAC. Consequently, adapter logic, TOEMACIF, is integrated between the TOE200GADV-IP and the F-Tile Ethernet Hard IP to facilitate the conversion of these interfaces. The TOEMACIF is an IP core provided in the TOE200GADV-IP suite.

The reference design incorporates three distinct clock domains: CpuClk for the CPU system, MacClk for interfacing with the F-Tile Ethernet Hard IP, and UserClk for the user logic of the TOE200GADV-IP. To facilitate asynchronous signal transfer between CpuClk and UserClk, AsyncAvlReg is specifically designed. Further details about each module within the TOE200GADVTest are provided below.

Note:

- 1) It is recommended to set the UserClk frequency of TOE200GADV-IP at 220 MHz or higher.
- 2) The MacClk frequency for the F-Tile Ethernet Hard IP is 402.832 MHz.

## 2.1 F-Tile Ethernet Hard IP

The F-Tile Ethernet Hard IP encompasses the MAC, PCS, and PMA layers, facilitating interfaces with external devices using 200G BASE-SR. This IP Core features a 512-bit MAC Segmented user interface operating at 402.832 MHz, necessitating adapter logic (TOEMACIF) for seamless connection to the TOE200GADV-IP. For more comprehensive information, please refer to “F-Tile Ethernet Hard IP User Guide” on the Intel website.

[https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interfa ce-protocols/agilex-f-tile-hard-ip.html](https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interfa-ce-protocols/agilex-f-tile-hard-ip.html)

In this reference design, the recommended settings for the Hard IP parameters are as follows.

### General Options

PMA type : FGT  
 Ethernet mode : 200GE-4  
 Client interface : MAC Segmented  
 FEC mode : IEEE802.3 RS(544,514) (CL 134)  
 PMA reference frequency : 156.25 MHz  
 System PLL frequency : 830.078125 MHz

### MAC Options

Tx maximum frame size : 9014  
 Rx maximum frame size : 9014

## 2.2 QSFPConfig

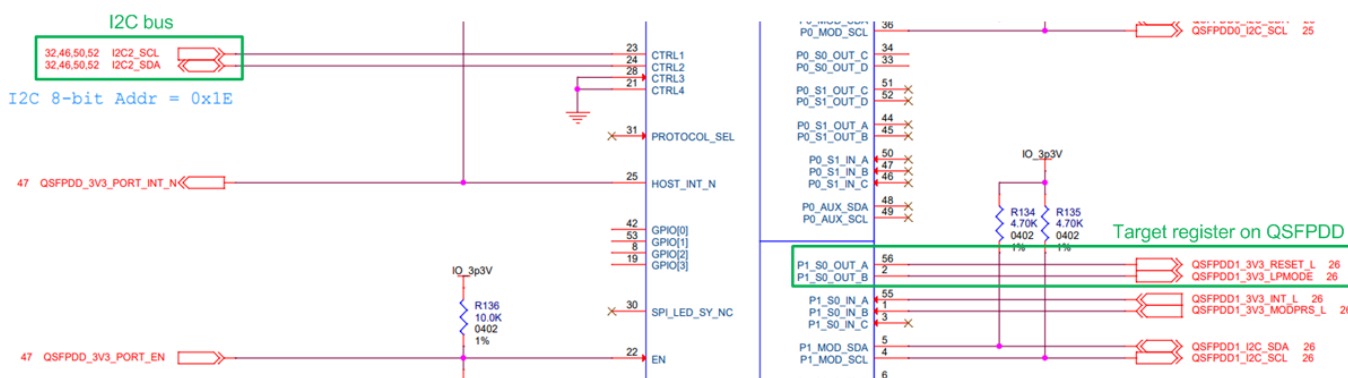


Figure 2-2 A part of QSFPDD IO schematic using in this reference design

On the Agilex 7 I-Series development kit, the ‘QSFP\_RESET\_L’ and ‘QSFP\_LPMODE’ signals on QSFPDD transceiver are configured using an I/O Expander module. This I/O Expander is controlled via two-line I<sup>2</sup>C bus protocol. Therefore, the QSFPConfig module is designed to serve as an I<sup>2</sup>C master, responsible for setting ‘QSFP\_RESET\_L’ to 1b and ‘QSFP\_LPMODE’ to 0b on the QSFPDD Transceiver. This configuration is necessary to ensure the proper operation and low-power mode management of the QSFPDD transceiver for high-speed data transfer in 200G Ethernet test environment.

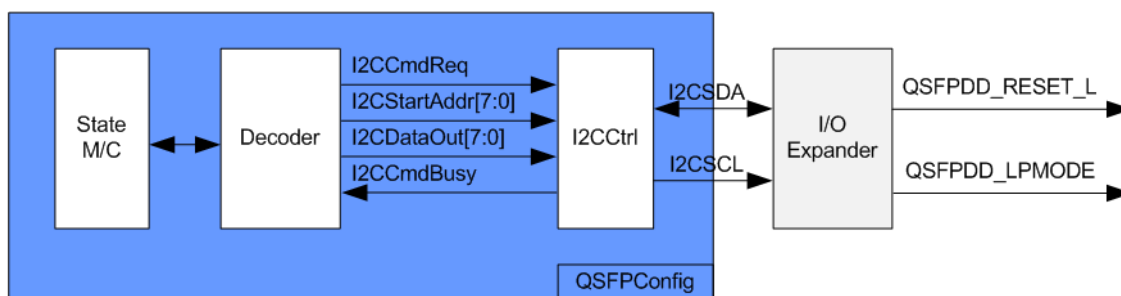


Figure 2-3 QSFPConfig block diagram

The QSFPConfig module includes the I2CCtrl module, which functions as the I<sup>2</sup>C master controller. The I2CCtrl module is responsible for writing one-byte data (I2CDataOut) to the specified register (I2CStartAddr) of the target device. Within the I2CCtrl logic, the device number and clock divider are preset, with the I2C clock (SCL) operating at 400 kHz. Within QSFPConfig, a state machine is designed to configure two registers of the I/O Expander:

- The OUT\_A and OUT\_B of Output Enable Register (Address=0x08)
- The OUT\_A and OUT\_B of Output Value Register (Address=0x0A)

Initially, the default settings of 'QSFP\_RESET\_L' and 'QSFP\_LPMODE', assigned to P1, are set to disabled (Hi-Z). This reference design necessitates enabling and setting toggling the output value of 'QSFP\_RESET\_L' to reset QSFPDD Transceiver. The state machine operates in the following sequences.

- 1) Enable Output: Sets the Output Enable Register of the P1 port to enable output by assigning I2CStartAddr=0x08 and I2CDataOut=0xFF (enable all outputs).
- 2) Reset QSFPDD: Sets the Output Value Register of the P1 port to set 'QSFP\_RESET\_L' to 0b by assigning I2CStartAddr=0x0A and I2CDataOut=0x00.
- 3) Release QSFPDD Reset: Sets the Output Value Register of the P1 port to set 'QSFP\_RESET\_L' to 1b by assigning I2CStartAddr=0x0A and I2CDataOut=0x0F.

Once the register configuration is completed, the QSFPConfig module transitions to the Idle state. Therefore, the configuration of P1 occurs once after the system is powered on. The timing diagram for the I2CCtrl user interface is shown in Figure 2-4.

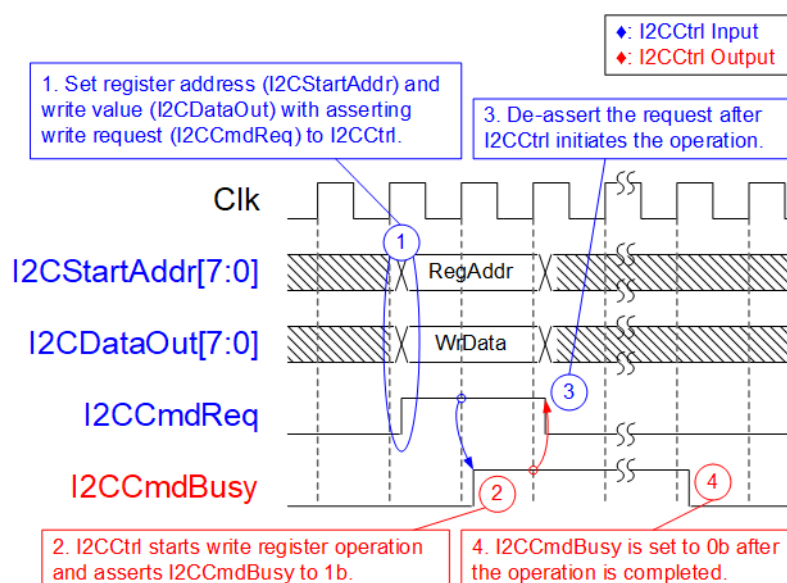


Figure 2-4 Timing diagram of I2CCtrl

- 1) I2CCmdReq is set to 1b with valid values for I2CStartAddr and I2CDataOut. All inputs must remain unchanged until I2CCtrl asserts I2CCmdBusy to 1b.
- 2) I2CCmdBusy is set to 1b after I2CCtrl has loaded all input parameters and initiated the write register operation.
- 3) I2CCmdReq is set to 0b to clear the write request and prepares the parameters for the next operation.
- 4) Once I2CCtrl completes the operation, I2CCmdBusy is de-asserted to 0b.

## 2.3 TOEMACIF

The TOEMACIF serves as adapter logic to seamlessly connect the TOE200GADV-IP with the F-Tile Ethernet Hard IP. It facilitates the interface conversion between the 1024-bit Avalon stream interface of the TOE200GADV-IP and the 512-bit MAC Segmented interface of F-Tile Ethernet Hard IP. Additionally, TOEMACIF is responsible for managing packet transfers across different clock domain between UserClk and MacClk. This adapter is provided as additional IP core, delivered with the TOE200GADV-IP. For further details about TOEMACIF, please refer to the TOE200GADV-IP datasheet available on our website.

[https://dgway.com/TOE-IP\\_A\\_E.html](https://dgway.com/TOE-IP_A_E.html)

## 2.4 AvISSw2to1

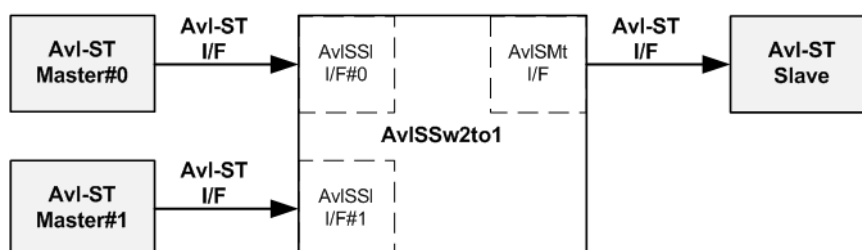


Figure 2-5 AvISSw2to1 interface

This module serves as a 2-to-1 switch logic for Avalon-ST interface, facilitating the transfer of transmitted data from User2MAC or TOE200GADV-IP to the F-Tile Ethernet Hard IP. AvISSw2to1 incorporates configurable parameters allowing the selection of data and empty signal widths. In this reference design, the data width for User2MAC and TOE200GADV-IP is set at 1024 bits, while the data width for the empty signal is 7 bits.

Conceptually, AvISSw2to1 operates by transferring data from two Masters (Ch#0 and Ch#1) to one Slave. In cases where both channels request data transfer simultaneously, AvISSw2to1 employs a priority mechanism, selecting the higher priority channel to initiate the data stream transfer until the end of the packet. Subsequently, the priority switches to the other channel, and the data stream of the second channel is transferred until the end of the packet.

The control signal 'rChSel' is employed by the AvISSw2to1 logic to select the active AvlSSI I/F, which is the interface connecting to the external Master. When two channels request data transfer while in an Idle condition, 'rChSel' changes its value to the new channel after completing the current channel's data transfer. Further details are illustrated in Figure 2-6.



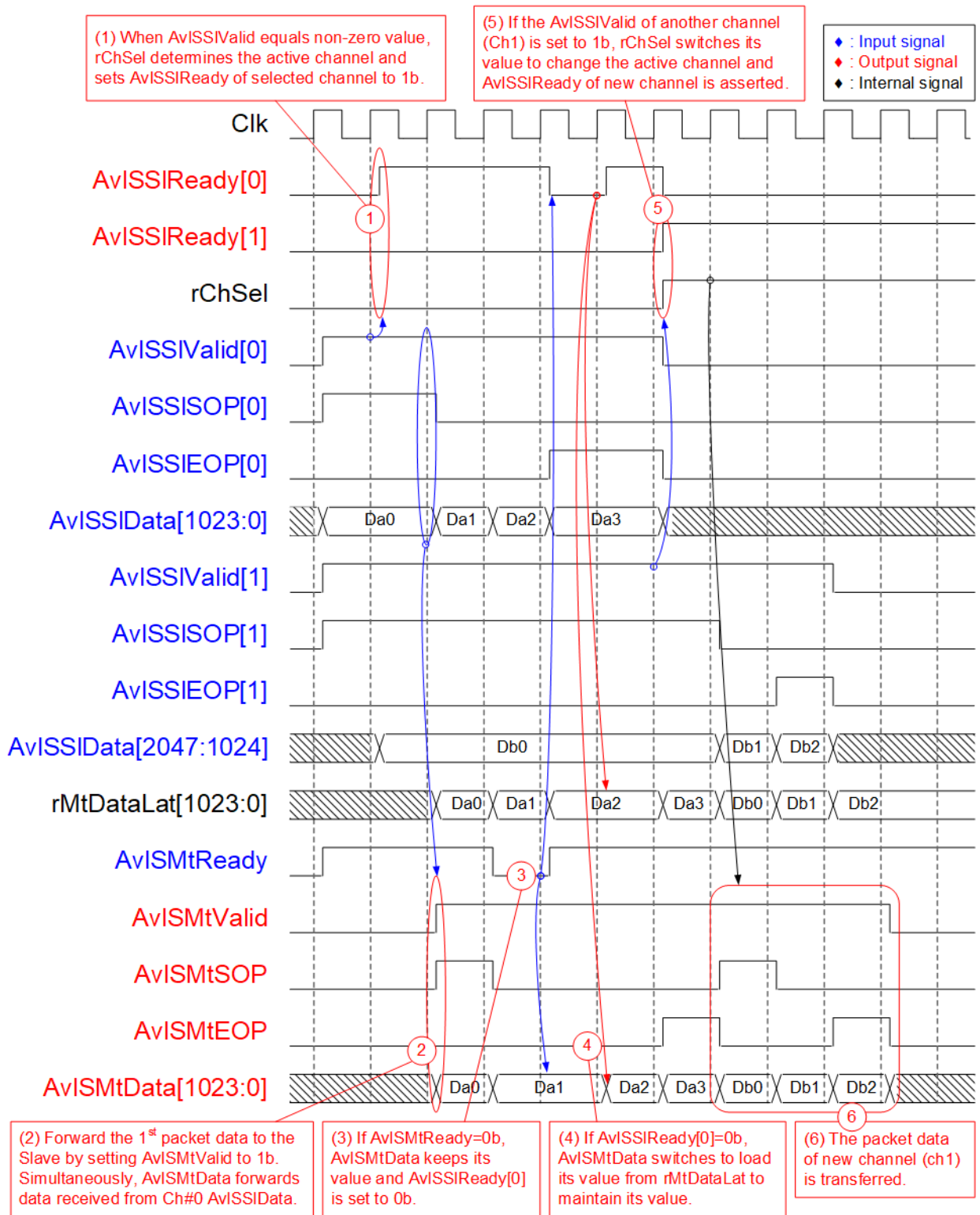


Figure 2-6 AvISSw2to1 timing diagram

- 1) When two users simultaneously initiate the transmission of a new packet by asserting AvISSIValid to 1b, and the module is currently in an Idle state, the value of rChSel (the signal indicating the active channel) remains unchanged to facilitate the forwarding of data from the same channel to the Slave. In Figure 2-6, Ch#0 is selected, prompting the assertion of AvISSIReady for the selected channel (Ch#0) to 1b, enabling the acceptance of the first data.
- 2) The input signals from the selected channel (Ch#0), including AvISSISOP[0] (indicating the start-of-packet), AvISSIEOP[0] (indicating the end-of-packet), and bits[1023:0] of AvISSIData (1024-bit data), are loaded as output signals to the external Slave via the Master I/F (AvISMtSOP, AvISMtEOP, and AvISMtData, respectively). Additionally, AvISMtValid is asserted to 1b, initiating the transmission of the new packet to the Slave.
- 3) When the Slave is not ready to receive data, indicated by the de-assertion of AvISMtReady to 0b, all output signals of the Master I/F maintain the same values. Also, AvISSIReady for the active channel is de-asserted to 0b, preserving the input signals from the Master.
- 4) Upon the Slave re-asserting AvISMtReady to accept data, the output signals to the Slave load the next values from the internal latch register (rMtDataLat). The internal latch register stores the latest value from the active source when AvISSIReady is asserted to 1b, ensuring the unsent data is stored and transmitted to the Slave when the Slave pauses data transmission.
- 5) After the final data of a packet from the active channel is accepted, the module scans for the next active channel. If AvISSIValid for another channel is asserted, rChSel switches its value. In Figure 2-6, the next active channel becomes Ch#1 (rChSel=1b), facilitating the acceptance of data from Ch#1.
- 6) The input signals (AvISSISOP[1], AvISSIEOP[1], and bits[2047:1024] of AvISSIData) from the active channel (Ch#1) are forwarded to become the output signals of the Slave (AvISMtSOP, AvISMtEOP, and AvISMtData) until the final data of packet is completely transferred.

## 2.5 TOE200GADV-IP

The TOE200GADV-IP implements TCP/IP offloading engine for handling four TCP sessions with the same target. The user data interface utilizes a 1024-bit Avalon stream interface. The control interface is used to configure the network parameters, send the command request, and monitor the operation status. The Ethernet MAC interface utilizes a 1024-bit Avalon stream interface. Further information of the IP can be found on our website.

[https://dgway.com/TOE-IP\\_A\\_E.html](https://dgway.com/TOE-IP_A_E.html)

## 2.6 User2MAC

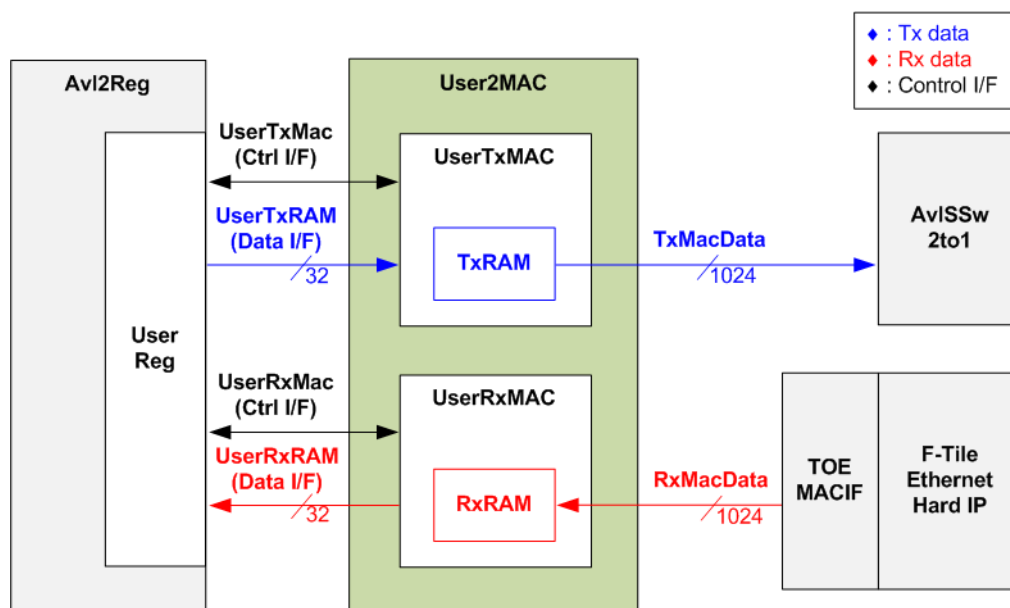


Figure 2-7 User2MAC block diagram

User2MAC is designed for transferring Ethernet packets in low-speed connection. The reference design incorporates the Ping command, using the ICMP protocol, to measure round-trip time. An ICMP echo reply packet generated upon receiving an ICMP echo request packet.

The CPU utilizes Avl2Reg to create and decode Ethernet packets, with a data bus width of 32 bits on the Avl2Reg side and 1024 bits on the MAC I/F side.

User2MAC operates in both transmission and reception modes, consisting of two modules: UserTxMAC and UserRxMAC. UserTxMAC includes TxRAM, where the CPU prepares and stores Ethernet packets to be transmitted. Meanwhile, UserRxMAC features RxRAM to store Ethernet packets received from the Ethernet MAC. Prior to storing a packet in RxRAM, a filtering logic checks the Ethernet header to ensure only valid packets are stored, while invalid ones are discarded. The CPU then reads from RxRAM to decode the stored packets.

Further details about the functionalities and operations of UserTxMAC and UserRxMAC are provided in the subsequent sections.

### 2.6.1 UserTxMac

UserTxMAC includes 32 x 1024-bit simple dual port RAM for storing packets to be transmitted. The CPU writes these packets to the RAM via the UserTxRam write I/F. The CPU sets the packet size (UserTxLen) and asserts a request (UserTxReq) to initiate the logic that forwards the packet, read from TxRAM, to the Ethernet MAC (EMAC). The transmit interface of the EMAC is a 1024-bit Avalon stream, which may de-assert its ready (TxReady) to temporarily pause data transmission. Upon completion of the packet transmission to EMAC, the busy signal (UserTxBusy) is de-asserted to 0b. Additional details about the internal logic design of UserTxMAC are illustrated in Figure 2-8.

*Note: The UserTxRam Write I/F with the CPU utilizes a 32-bit data width, while the TxRAM data width is 1024 bits. Therefore, a decoder is implemented to create a write byte enable signal, allowing the CPU to write specific bytes of the 1024-bit data bus of TxRAM.*

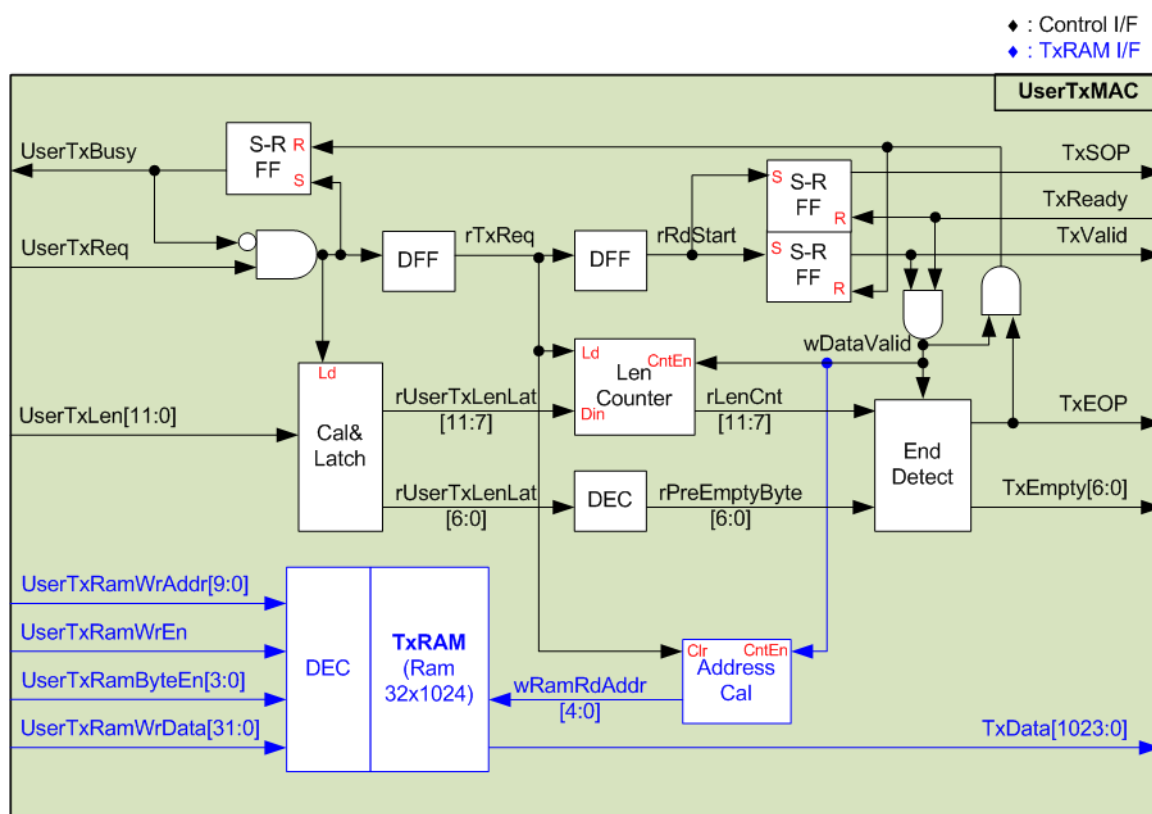


Figure 2-8 UserTxMAC Logic Diagram

The steps involved in transmitting a packet from UserTxMAC are outlined below.

- 1) The CPU verifies that UserTxBusy is 0b to ensure that UserTxMAC is in an idle state.
- 2) The CPU prepares the packet and writes it to TxRAM, starting at address 0 (UserTxRamWrAddr=0). The maximum size of the transmitted packet is 4 KB, matching the size of TxRAM.  
*Note: TxRAM incorporates a byte enable feature, allowing the CPU to write data in byte unit.*
- 3) The CPU sets UserTxLen to specify the transmit packet size in byte units and asserts UserTxReq to 1b to initiate data transmission.
- 4) Subsequently, the request signal is fed to several logics and a DFF chain. UserTxBusy is asserted to 1b, indicating that the operation is in progress. The total transfer size (UserTxLen) is loaded into internal logic and split into two parts.
  - UserTxLen[11:7] determines the amount of 1024-bit data, rounded up if not aligned.
  - UserTxLen[6:0] is latched to create the empty byte for the final packet data (rPreEmptyByte and TxEmpty) using a decoder.
- 5) When the start flag (rRdStart) is asserted, the first data is read from TxRAM, and the data valid (TxValid) is asserted to 1b with the start of packet flag (TxSOP). The read address (wRamRdAddr) is incremented to fetch the next data after each transfer (TxValid=1b and TxReady=1b). Additionally, the Length counter (rLenCnt) is decremented after each data transfer to track the position of the final data.
- 6) When rLenCnt=1 or the next data is the final data, the end of packet flag (TxEOP) is asserted to 1b. Also, the empty byte (TxEmpty) loads the value from rPreEmptyByte. TxEmpty is set to all zeroes for 1024-bit data if it is not the final data.
- 7) After the final data is transferred, UserTxBusy is de-asserted to 0b, indicating the completion of the current transfer.

## 2.6.2 UserRxMAC

UserRxMAC performs three distinct operations to validate received packets and store valid packets in RxRAM, which is a 32 x 1024-bit simple dual port RAM. Accordingly, the logic within UserRxMAC is categorized into three groups, as shown in Figure 2-9.

*Note: The UserRxRam Read I/F with the CPU employs a 32-bit data width, while RxRAM has a data width of 1024 bits. Therefore, a 32-to-1 Mux is integrated to select 32-bit data from the 1024-bit data.*

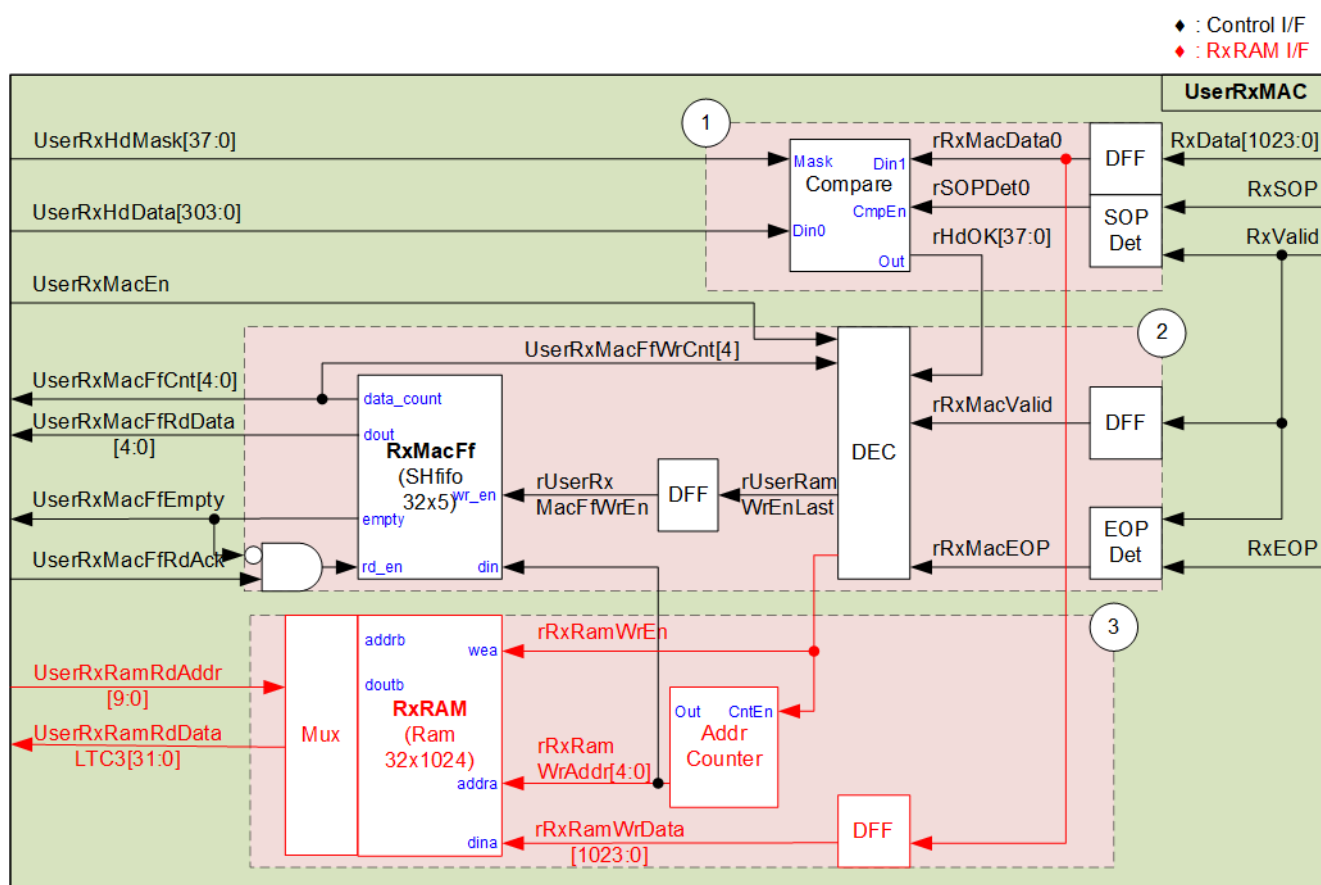


Figure 2-9 UserRxMAC Logic Diagram

Block (1) comprises the logic responsible for verifying the 38-byte header (byte#0 – byte#37) of each received packet. The user can set the expected values and mask bits for data comparison. Only packets with the correct header are accepted.

Block (2) check the enable flag from the user and the available space in RxMacFf. If the module is disabled by the user or if the FIFO lacks sufficient space, the packet will be rejected. RxMacFf stores the end address of RxRAM after a packet is stored, allowing the CPU to determine the received packet size based on this end address.

Block (3) is dedicated to storing the received packet in RxRAM. The details of UserRxMAC operation upon receiving a packet are elaborated as follows.

- 1) Header Verification: Two user-configured parameters, 38-byte header data (UserRxHdData) and a 38-bit data mask for verifying the packet header (UserRxHdMask), must remain stable when the user enables this module by asserting UserRxMacEn to 1b. Upon receiving the first data of a new packet, SOPDet asserts rSOPDet0 to initiate the Compare module. The 38-byte header data is compared to byte#0 - byte#37 of the received data, controlled by the data mask bit. Each bit of the data mask corresponds to one byte of received data. If the data mask is de-asserted to 0b, that data byte is bypassed. Therefore, header verification is disabled if UserRxHdMask is set to all zeros. When a specific byte of the received packet header is valid, the dedicated bit of rHdOK is asserted to 1b. The packet can be stored in RxRAM only when all 38 bits of rHdOK are set to 1.
- 2) Enable and Free Space Check: Two signals are read – the enable flag from the user (UserRxMacEn) and the RxMacFf data counter (UserRxMacFfWrCnt). It confirms that the CPU is ready to process the received packet by asserting UserRxMacEn to 1b and this module has sufficient free space to store the received packet and the write pointer of RxRAM. Bit[4] of UserRxMacFfWrCnt must be equal to 0b. If both conditions are met and the header is valid, the write enable of RxRAM (rRxRamWrEn) is asserted to store the received data in RxRAM. RxMacFf stores the RAM address after storing each packet in RxRAM. Therefore, EOPDet asserts a pulse of rRxRamWrEnLast when the end of the packet is received. After that, rUserRxMacFfWrEn is asserted to 1b to write the end address to RxMacFf.
- 3) Packet Storage: The valid packet is stored in RxRAM by asserting the write enable to RxRAM (rRxRamWrEn) when the data is received (RxValid=1b). The address counter is incremented after each 1024-bit data is stored in RxRAM. The last address after receiving the end-of-packet is stored in RxMacFf.

*Note: Using bit[4] of UserRxMacFfWrCnt for checking FIFO space, up to 16 addresses can be stored in RxMacFf. Therefore, up to 16 packets can be stored in RxRAM. Given that the RxRAM size is 4 KB, one packet size should not exceed 256 bytes. However, users have the flexibility to adjust RAM size and FIFO size to align with their system requirements.*

The CPU's process for handling received packets stored in UserRxMAC is outlined as follows.

- 1) The CPU awaits the condition where the FIFO is not empty (UserRxMacFfEmpty=0b).
- 2) The CPU reads the last address using the UserRxMacFfRdData signal, which is valid for reading due to RxMacFf being a show-ahead FIFO.
- 3) After that, the CPU asserts UserRxMacFfRdAck to 1b to flush the read data from RxMacFf.
- 4) The CPU reads and decodes a received packet from RxRAM, starting from the latest read position to the last address obtained from RxMacFf. Upon completing packet processing, the 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 1024-bit data. Therefore, the CPU firmware must convert the 1024-bit address stored in RxMacFf to a 32-bit address before starting data reading from RxRAM.*

## 2.7 CPU and Peripherals

The CPU system uses a 32-bit Avalon-MM bus as the interface to access peripherals such as the Timer and JTAG UART. The system also integrates an additional peripheral to access the test logic by assigning a unique base address and address range. To support CPU read and write operations, the hardware logic must comply with the Avalon-MM bus standard. Avl2Reg module, as shown in Figure 2-10, is designed to connect the CPU system via the Avalon-MM interface, in compliance with the standard.

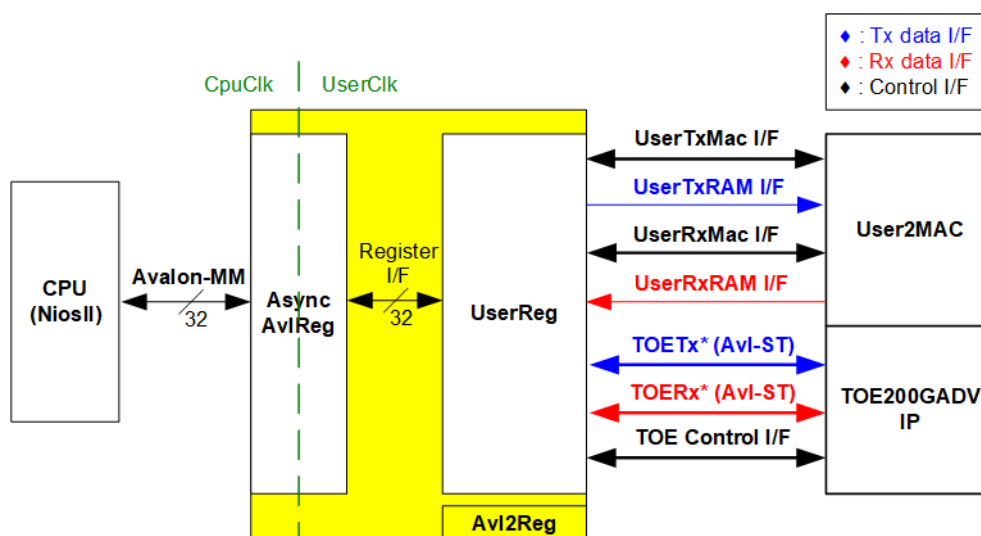


Figure 2-10 Avl2Reg block diagram

Avl2Reg consists of AsyncAvlReg and UserReg. AsyncAvlReg converts Avalon-MM signals into a simple Register interface with a 32-bit data bus size, similar to Avalon-MM data bus size. It also includes asynchronous logic to handle clock domain crossing between the CpuClk and UserClk domains.

UserReg includes the Register files designed to store parameters and status signals for both User2MAC (via UserTxMac I/F and UserRxMac I/F) and TOE200GADV-IP (via TOE Control I/F). The data interface of User2MAC utilizes a simple dual-port RAM interface, which aligns with the Register I/F. While the data interface of TOE200GADV-IP employs an Avl-ST interface, facilitated through TOETx I/F and TOERx I/F. Additional details regarding AsyncAvlReg and UserReg are provided below.



### 2.7.1 AsyncAviReg

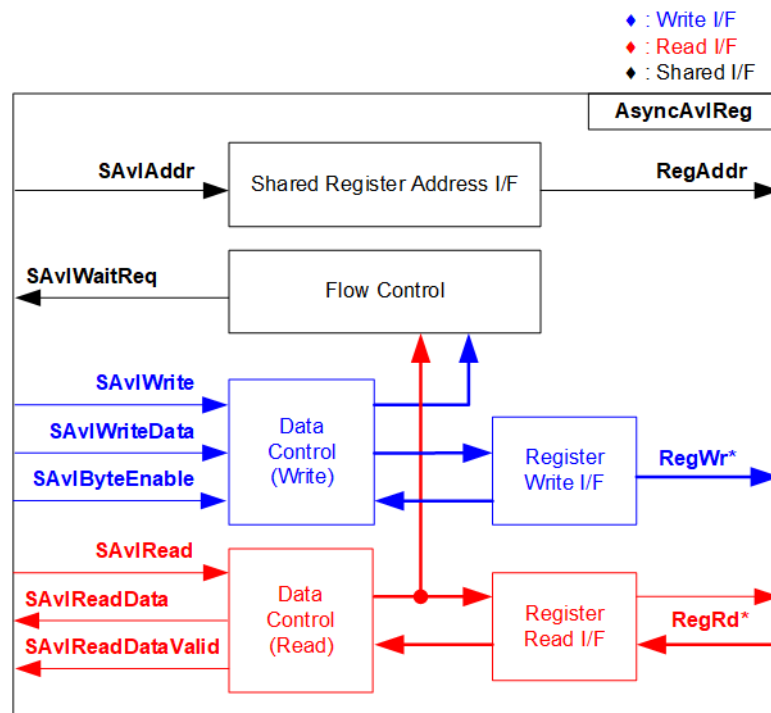


Figure 2-11 AsyncAviReg interface

The Avalon-MM bus interface signal can be grouped into three categories: Write channel (blue), Read channel (red), and Shared control channel (black). More details about the Avalon-MM interface specification can be found in the following document.

[https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl\\_a\\_valon\\_spec.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_a_valon_spec.pdf)

According to Avalon-MM specification, only one command (write or read) can be executed at a time. AsyncAviReg’s logic is divided into three groups: Write control logic, Read control logic, and Flow control logic. The flow control logic asserts SAviWaitReq to hold the next request from the Avalon-MM interface if the current request has not finished. Write control and Write data I/F of the Avalon-MM bus are latched and transferred to the Write register interface with clock domain crossing registers. Similarly, Read control I/F are latched and transferred to be Read register interface. Afterward, the data returned from Register Read I/F is transferred to Avalon-MM bus with using clock domain crossing registers. The Address I/F of Avalon-MM is also latched and transferred to the Address register interface.

The Register interface is compatible with the single-port RAM interface for write transactions. However, the read transaction of the Register interface is slightly modified from RAM interface by adding RdReq and RdValid signals to control the read latency time. Since the address of the Register interface is shared for write and read transactions, the user cannot write and read the register at the same time. The timing diagram of the Register interface is shown in Figure 2-12.

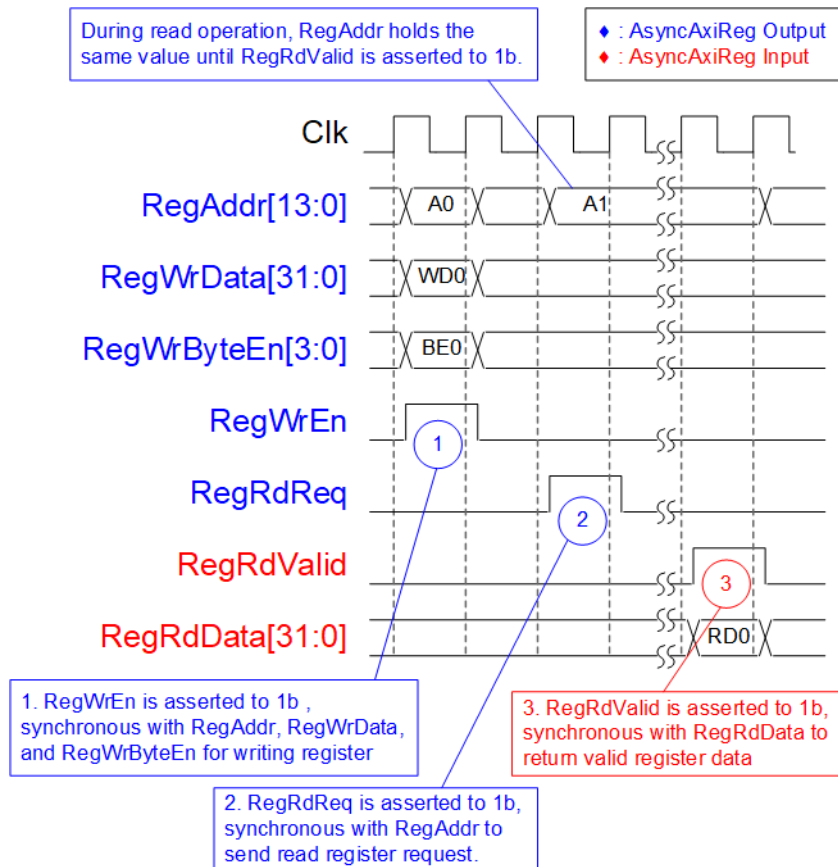


Figure 2-12 Register interface timing diagram

- 1) Timing diagram to write register is similar to that of a single-port RAM. The RegWrEn signal is set to 1b, along with a valid RegAddr (Register address in 32-bit units), RegWrData (write data for the register), and RegWrByteEn (write byte enable). The byte enable consists of four bits that indicate the validity of the byte data. For example, bit[0], [1], [2], and [3] are set to 1b when RegWrData[7:0], [15:8], [23:16], and [31:24] are valid, respectively.
- 2) To read register, AsyncAvlReg sets the RegRdReq signal to 1b with a valid value for RegAddr. The 32-bit data is returned after the read request is received. The slave detects the RegRdReq signal being set to start the read transaction. In the read operation, the address value (RegAddr) remains unchanged until RegRdValid is set to 1b. The address can then be used to select the returned data using multiple layers of multiplexers.
- 3) The slave returns the read data on RegRdData bus by setting the RegRdValid signal to 1b. After that, AsyncAvlReg forwards the read value to the SAvlRead interface.

### 2.7.2 UserReg

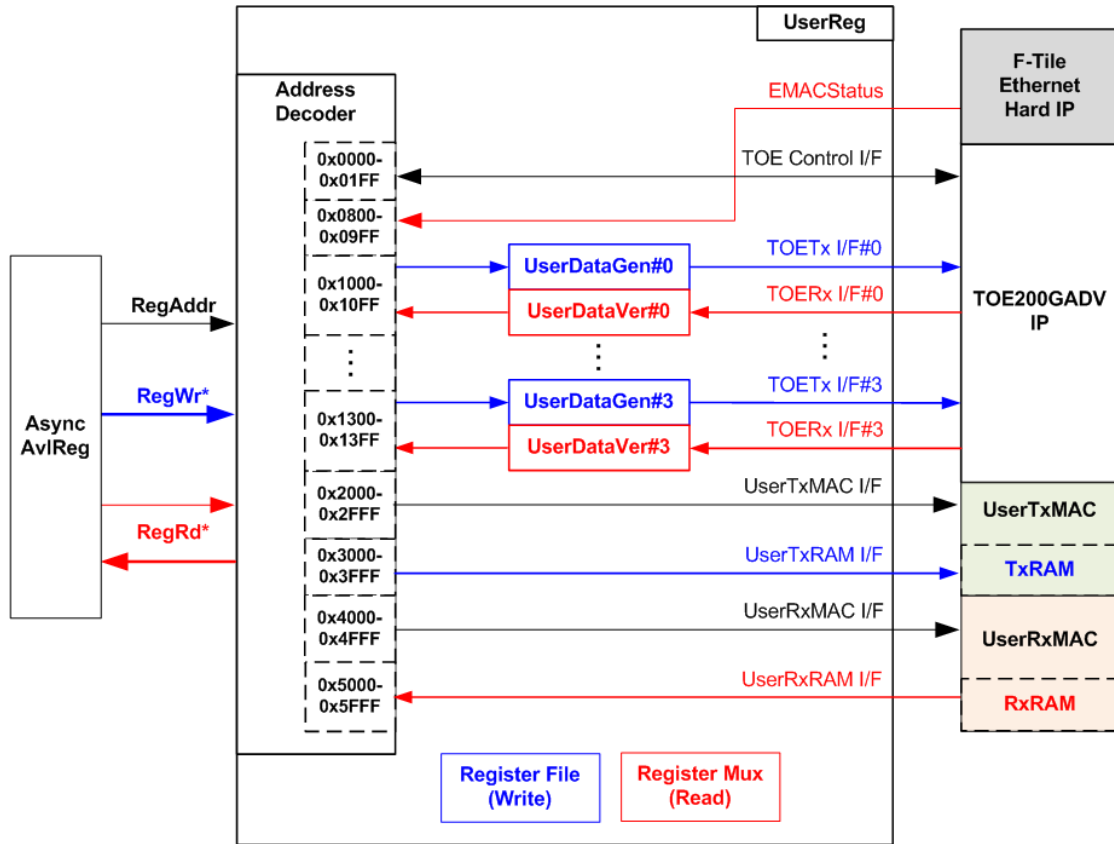


Figure 2-13 UserReg block diagram

UserReg implements three key operations: an Address decoder with a Register File for write access and a Register Mux for read access, User Data Generator (UserDataGen), and User Data Verification (UserDataVer). Detailed information is provided below.

#### Address decoder with Register File and Register Mux

As shown in Figure 2-13, the address range mapped to UserReg is divided into seven areas.

- 1) 0x0000 – 0x01FF: Control and status signals of TOE200GADV-IP
- 2) 0x0800 – 0x09FF: Status signal of EMAC
- 3) 0x1000 – 0x1013FF: UserDataGen and UserDataVer signals for transferring data with 4 sessions of TOE200GADV-IP
- 4) 0x2000 – 0x2FFF: Control and status signals of UserTxMAC
- 5) 0x3000 – 0x3FFF: Write interface of Tx RAM inside UserTxMAC
- 6) 0x4000 – 0x4FFF: Control and status signals of UserRxMAC
- 7) 0x5000 – 0x5FFF: Read interface of Rx RAM inside UserRxMAC

The upper bits of RegAddr are utilized 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 outlined Table 2-1.

To read register, multiple multiplexers are utilized to select data from each module, leading to increased read latency time due to the multiplexer. The slowest path for returning read data is from UserRxRAM, which has a latency time of four clock cycles. Therefore, RegRdValid is generated by RegRdReq through the assertion of four D Flip-flops

**Table 2-1 Register map Definition**

Address	Register Name	Description
Wr/Rd	(Label in the "toe200gadvtest.c")	
<b>(BA+0x0000) – (BA+0x01FF): Control and Status of TOE200GADV-IP</b>		
<b>Further information of each TOE200GADV-IP I/O signals is described in the datasheet.</b>		
<b>(BA+0x0000) – (BA+0x00FF): Hardware system signals</b>		
BA+0x0000	Hardware reset	[0]: Set to 1b to reset TOE200GADV-IP, all UserDataGen modules, and all UserDataVer modules.
Wr/Rd	(HW_RST_INTREG)	
BA+0x0084	Ethernet MAC status	[0]: Ethernet MAC link status (0b-Link down, 1b-Link up)
Rd	(EMAC_STS_INTREG)	[6]: Rx alignment status (0b-Not aligned, 1b-Aligned) [7]: RxPCS ready status (0b-RxPCS is busy, 1b-RXPCS is ready) [8]: Remote Fault Code detect (0b-Not detect, 1b-Detect)
<b>(BA+0x0100) – (BA+0x013F): Common parameters and status signals</b>		
BA+0x0100	TOE IP version	[31:0]: Mapped to IPVersion[31:0] of TOE200GADV-IP
Rd	(TOE_VER_INTREG)	
BA+0x0104	TOE initial finish flag	[0]: Mapped to InitFinish of TOE200GADV-IP
Rd	(TOE_INF_INTREG)	
BA+0x0110	Source MAC Address Low	[31:0]: Mapped to SrcMacAddr[31:0] of TOE200GADV-IP
Wr/Rd	(TOE_SML_INTREG)	
BA+0x0114	Source MAC Address High	[15:0]: Mapped to SrcMacAddr[47:32] of TOE200GADV-IP
Wr/Rd	(TOE_SMH_INTREG)	
BA+0x0118	Dest MAC Address In Low	[31:0]: Mapped to DstMacAddr[31:0] of TOE200GADV-IP
Wr/Rd	(TOE_DMIL_INTREG)	
BA+0x011C	Dest MAC Address In High	[15:0]: Mapped to DstMacAddr[47:32] of TOE200GADV-IP
Wr/Rd	(TOE_DMIH_INTREG)	
BA+0x0120	Source IP Address	[31:0]: Mapped to SrcIPAddr[31:0] of TOE200GADV-IP
Wr/Rd	(TOE_SIP_INTREG)	
BA+0x0124	Dest IP Address	[31:0]: Mapped to DstIPAddr[31:0] of TOE200GADV-IP
Wr/Rd	(TOE_DIP_INTREG)	
BA+0x0128	Dest MAC Mode	[1:0]: Mapped to DstMacMode[1:0] of TOE200GADV-IP
Wr/Rd	(TOE_DMM_INTREG)	
BA+0x012C	Window Threshold	[9:0]: Mapped to WindowThres[9:0] of TOE200GADV-IP
Wr/Rd	(TOE_WIN_INTREG)	
BA+0x0130	TCP Control Timeout	[31:0]: Mapped to TCPCtlTimeOutSet[31:0] of TOE200GADV-IP
Wr/Rd	(TOE_TCT_INTREG)	
BA+0x0134	TCP Receive Timeout	[23:0]: Mapped to TCPRxTimeOutSet[23:0] of TOE200GADV-IP
Wr/Rd	(TOE_TRT_INTREG)	
BA+0x0138	Dest MAC Address Out Low	[31:0]: Mapped to DstMacAddrOut [31:0] of TOE200GADV-IP
Rd	(TOE_DMOL_INTREG)	
BA+0x013C	Dest MAC Address Out High	[15:0]: Mapped to DstMacAddrOut [47:32] of TOE200GADV-IP
Rd	(TOE_DMOH_INTREG)	

Address	Register Name	Description
Wr/Rd	(Label in the "toe200gadvtest.c")	
<b>(BA+0x0140) – (BA+0x017F): Session control and status signals</b>		
BA+0x0140	Source Port Number	[15:0]: Mapped to TCPSrcPort[15:0] of TOE200GADV-IP
Wr/Rd	(TOE_SPN_INTREG)	
BA+0x0144	Dest Port Number	[15:0]: Mapped to TCPDstPort[15:0] of TOE200GADV-IP
Wr/Rd	(TOE_SPN_INTREG)	
BA+0x0148	TCP Last Mode	[1:0]: Mapped to TCPLastMode[1:0] of TOE200GADV-IP
Wr/Rd	(TOE_LMD_INTREG)	
BA+0x014C	TCP Command	Wr [3:0]: Set value to TCPConnCmd[3:0] of TOE200GADV-IP When this register is written, the connection request (TCPConnReq) is asserted to initiate the TOE200GADV-IP operation. Rd [3:0]: The latest value of TCPConnCmd[3:0] set to TOE200GADV-IP [8]: Mapped to TCPConnReady of TOE200GADV-IP
BA+0x0150	TCP Connection ON	[3:0]: Mapped to TCPConnOn[3:0] of TOE200GADV-IP
Rd	(TOE_CON_INTREG)	
BA+0x0154	TOE Interrupt	Wr - Set the specified bit to 1b to clear the corresponding interrupt, read from this register. For instance, if bit[0] of TOE_INT_INTREG is set to 1b to indicate retry interrupt from common functions, users can set bit[0] of TOE_INT_INTREG to 1b to clear the interrupt, and the read value of this bit will become zero value. Rd – The interrupt status is activated by various conditions. [0]: Set to 1b when TCPRtrInt is triggered by common functions (TCPRtrIntStatus[15:0] is non-zero). [8]: Set to 1b when TCPConnCpl is triggered by session#0 commands (TCPConnCplStatus[3:2] is 00b). [9]: Set to 1b when TCPConnCpl is triggered by session#1 commands (TCPConnCplStatus[3:2] is 01b). [10]: Set to 1b when TCPConnCpl is triggered by session#2 commands (TCPConnCplStatus[3:2] is 10b). [11]: Set to 1b when TCPConnCpl is triggered by session#3 commands (TCPConnCplStatus[3:2] is 11b). [16]: Set to 1b when TCPRtrInt is triggered by session#0 functions (TCPRtrIntStatus[31:16] is non-zero). [17]: Set to 1b when TCPRtrInt is triggered by session#1 functions (TCPRtrIntStatus[47:32] is non-zero). [18]: Set to 1b when TCPRtrInt is triggered by session#2 functions (TCPRtrIntStatus[63:48] is non-zero). [19]: Set to 1b when TCPRtrInt is triggered by session#3 functions (TCPRtrIntStatus[79:64] is non-zero). [24]: Set to 1b when TCPRstInt is triggered by session#0 functions (TCPRstIntStatus[31:16] is non-zero). [25]: Set to 1b when TCPRstInt is triggered by session#1 functions (TCPRstIntStatus[47:32] is non-zero). [26]: Set to 1b when TCPRstInt is triggered by session#2 functions (TCPRstIntStatus[63:48] is non-zero). [27]: Set to 1b when TCPRstInt is triggered by session#3 functions (TCPRstIntStatus[79:64] is non-zero).
Wr/Rd	(TOE_INT_INTREG)	

Address Wr/Rd	Register Name (Label in the "toe200gadvtest.c")	Description
<b>(BA+0x0180) – (BA+0x01FF): Session#0 – Session#3 status signals</b>		
BA+0x0180 Rd	TCP Conn Status#0 Low (TOE_TCS0L_INTREG)	[31:0]: Mapped to TCPConnStatus[31:0] of TOE200GADV-IP (session#0)
BA+0x0184 Rd	TCP Conn Status#0 High (TOE_TCS0H_INTREG)	[31:0]: Mapped to TCPConnStatus[63:32] of TOE200GADV-IP (session#0)
BA+0x0188 Rd	TOE Transmit Status#0 (TOE_TTS0_INTREG)	[31:0]: Mapped to TOETxStat0[31:0] of TOE200GADV-IP (session#0 status)
BA+0x018C Rd	TOE Receive Status#0 (TOE_TRS0_INTREG)	[31:0]: Mapped to TOERxStat0[31:0] of TOE200GADV-IP (session#0 status)
BA+0x0190 Rd	Conn Completion Status#0 (TOE_CCS0_INTREG)	[4:0]: Latched value of TCPConnCplStatus[4:0] when TCPConnCpl is triggered by session#0 commands.
BA+0x0194 Rd	Retry Interrupt Status#0 (TOE_RTS0_INTREG)	[15:0]: Latched value of TCPRtrIntStatus[31:16] when TCPRtrInt is triggered by session#0 functions.
BA+0x0198 Rd	Reset Interrupt Status#0 (TOE_RSS0_INTREG)	[15:0]: Latched value of TCPRstIntStatus[31:16] when TCPRstInt is triggered by session#0 functions.
BA+0x01A0 – BA+0x01BB	TOE_TCS1L_INTREG – TOE_RSS1_INTREG	Similar to (BA+0x0180) – (BA+0x019B), these registers indicate the status of session#1.
BA+0x01C0 – BA+0x01DB	TOE_TCS2L_INTREG – TOE_RSS2_INTREG	Similar to (BA+0x0180) – (BA+0x019B), these registers indicate the status of session#2.
BA+0x01E0 – BA+0x01FB	TOE_TCS3L_INTREG – TOE_RSS3_INTREG	Similar to (BA+0x0180) – (BA+0x019B), these registers indicate the status of session#3.
<b>(BA+0x1000) – (BA+0x13FF): UserDataGen and UserDataVer interface</b>		
<b>(BA+0x1000) – (BA+0x107F): UserDataGen#0 control/status</b>		
BA+0x1000 Wr/Rd	User#0 Transmit Command (USR0TX_CMD_INTREG)	Wr [0]: Set to 1b to send the request to UserDataGen#0, triggering data sending function. This signal is auto-cleared after initiating data transmission. [1]: Set to 1b to clear the value read from USR0TX_LEN/H_INTREG. Rd[0]: Indicate busy flag of UserDataGen#0. 0b-Idle, 1b-Data is transmitting.
BA+0x1004 Wr/Rd	User#0 Tx Transfer Speed (USR0TX_TRS_INTREG)	[6:0]: Set maximum performance for transmitting data in percentage unit of 32,000 MB/s (250 MHz x 1024-bit data). Valid range is 1 – 100.
BA+0x1008 Wr/Rd	User#0 Transmit Length Low (USR0TX_LEN_L_INTREG)	Wr [31:0]: Bits[31:0] of total transmit size in byte unit. Rd [31:0]: Bits[31:0] of complete transmit size in byte unit
BA+0x100C Wr/Rd	User#0 Transmit Length High (USR0TX_LEN_H_INTREG)	Wr [15:0]: Bits[47:32] of total transmit size in byte unit. Rd [15:0]: Bits[47:32] of complete transmit size in byte unit
BA+0x1010 Wr/Rd	User#0 Tx Packet Len Low (USR0TX_PKLL_INTREG)	[31:0]: Bits[31:0] of transmit packet size in byte unit.
BA+0x1014 Wr/Rd	User#0 Tx Packet Len High (USR0TX_PKLH_INTREG)	[15:0]: Bits[47:32] of transmit packet size in byte unit.
<b>(BA+0x1080) – (BA+0x10FF): UserDataVer#0 control/status</b>		
BA+0x1080 Wr/Rd	User#0 Receive Command (USR0RX_CMD_INTREG)	Wr [0]: Set to 1b to enable receive function of UserDataVer#0. 0b-Disable receive function, 1b-Enable receive function. [1]: Set to 1b to enable data verification function of UserDataVer#0. 0b-Disable verification function, 1b-Enable verification function. [2]: Set to 1b to clear the value read from USR0RX_LEN/H_INTREG. Rd [0]: Indicate verification error status. 0b-No error, 1b-Verification is error.
BA+0x1084 Wr/Rd	User#0 Recv Transfer Speed (USR0RX_TRS_INTREG)	[6:0]: Set maximum performance for receiving data in percentage unit of 32,000 MB/s (250 MHz x 1024-bit data). Valid range is 1 – 100.
BA+0x1088 Rd	User#0 Receive Length Low (USR0RX_LEN_L_INTREG)	Rd [31:0]: Bits[31:0] of total receive size in byte unit
BA+0x108C Rd	User#0 Receive Length High (USR0RX_LEN_H_INTREG)	Rd [15:0]: Bits[47:32] of total receive size in byte unit

Address	Register Name	Description
Wr/Rd	(Label in the "toe200gadvtest.c")	
<b>BA+0x1100 – BA+0x13FF: UserDataGen and UserDataVer interface for session#1 – session #3</b>		
BA+0x1100 – BA+0x11FF		Similar to (BA+0x1000) – (BA+0x10FF), these registers are mapped to the control/status signals of UserDataGen#1 and UserDataVer#1 for session#1.
BA+0x1200 – BA+0x12FF		Similar to (BA+0x1000) – (BA+0x10FF), these registers are mapped to the control/status signals of UserDataGen#2 and UserDataVer#2 for session#2.
BA+0x1300 – BA+0x13FF		Similar to (BA+0x1000) – (BA+0x10FF), these registers are mapped to the control/status signals of UserDataGen#3 and UserDataVer#3 for session#3.
<b>BA+0x2000 – BA+0x3FFF: UserTxMAC</b>		
BA+0x2000	UserTxMAC Transmit Length	Wr [11:0]: Total amount of transmitted data in byte unit. Valid from 1 – 4095.
Wr/Rd	(TXEMAC_LEN_INTREG)	After this register is written, UserTxMAC initiates data transmission to EMAC. Rd [0]: Indicate busy status of UserTxMAC. 0b-Idle, 1b-Packet is transmitting.
BA+0x3000 – BA+0x3FFF	TxRAM in UserTxMAC	TxRAM area for storing transmitted packet, created by CPU, for low-speed connection.
Wr	(TXRAM_BASE_ADDR)	
<b>BA+0x4000 – BA+0x5FFF: UserRxMAC</b>		
BA+0x4000 – BA+0x4027	UserRxMAC Header Data	38 bytes of header data is utilized for packet filtering within UserRxMAC. This facilitates packet header comparison from byte#0 to byte#37 in each received packet. To activate the packet filtering logic, the user must additionally set RXEMAC_CMD_INTREG[0] to 1b, enabling the receive operation.
Wr/Rd	(RXEMAC_HDVAL_ADDR)	The byte mappings for header data are as follows. 0x4000[7:0], [15:8], [23:16], [31:24] correspond to byte#0, #1, #2, #3. 0x4004[7:0], [15:8], [23:16], [31:24] correspond to byte#4, #5, #6, #7. ... 0x4020[7:0], [15:8], [23:16], [31:24] correspond to byte#32, #33, #34. 0x4024[7:0], [15:8] correspond to byte#36, #37.
BA+0x4040 – BA+0x4047	UserRxMAC Header Byte Enable	A 38-bit signal is used to activate the verification function for 38-byte data, with each bit dedicated to controlling individual byte data.
Wr/Rd	(RXEMAC_HDEN_ADDR)	Byte-wise mappings for the signal are as follows. 0x4040[0], [1], [2], ..., [31] correspond to byte#0, #1, #2, ..., #31. 0x4044[0], [1], [2], ..., [5] correspond to byte#32, #33, #34, ..., #37. The states of each bit are defined as follows. 0b-Disable byte filtering (bypass data), 1b: Enable byte filtering.
BA+0x4060	UserRxMAC Command	Wr/Rd
Wr/Rd	(RXEMAC_CMD_INTREG)	[0] – Set to 1b to enable UserRxMAC module. 0b-Disable, 1b-Enable. Wr [1] – Set to 1b to assert the Read enable of RxMacFf within UserRxMAC module. Since RxMacFf is Show-Ahead type, writing to this register is used to flush one existing data from the FIFO. The data output of RxMacFf can be read from bits[4:0] of RXMAC_FF_INTREG. The user sets this bit to 1b once to flush one data from FIFO.
BA+0x4064	RxMacFf of UserRxMAC	[4:0]: Mapped to read data output from RxMacFf.
Rd	(RXEMAC_FF_INTREG)	[15]: Indicate the empty status of RxMacFf.
BA+0x5000 – BA+0x5FFF	RxRAM in UserRxMAC	RxRAM area for storing received packet. To process the received packet from a low-speed connection, the CPU accesses and decodes the packets from the RxRAM.
Rd	(RXRAM_BASE_ADDR)	

## User Data Generator

Within the UserReg module, the UserDataGen submodule functions as the user logic responsible for facilitating high-speed data transmission to TOE200GADV-IP. Each UserDataGen instance is dedicated to transmitting data for one TCP session data to TOE200GADV-IP. This reference design incorporates the utilization of four UserDataGen modules.

To initiate the test operation of this module, users trigger the start signal, TrnStart, setting it to 1b. This action, accompanied by specifying parameters such as Total data transfer size (TotDataLenSet) in bytes, Packet size (PacketSizeSet) in bytes, and Maximum speed (MaxSpeed) as a percentage (ranging from 1 to 100), kicks off the data transmission process.

If the value of 'TotDataLenSet' surpasses that of 'PacketSizeSet', UserDataGen generates a data stream using multiple packets. The transmission of the first data of the next packet occurs in the following clock cycle immediately after sending the last data of the preceding packet, without any pause. However, the MaxSpeed parameter is utilized to control the pause cycle for transmitting data every 100 transfer cycles. For instance, if 'MaxSpeed' is set to 90, a pause time of 10 clock cycles is inserted every 100 transfer cycles to control the speed of data transfer.

Upon the assertion of TrnStart, the busy flag (TrnBusy) is set to 1b, indicating that data transmission is in progress. Users can track the transmission progress by reading the completed transfer size (CurTrnSize) in bytes. This information can be reset by the user through the 'ClrTrnSize' signal or by initiating a new transfer request via 'TrnStart'. The data stream is generated and transmitted through a 1024-bit Avalon stream interface.

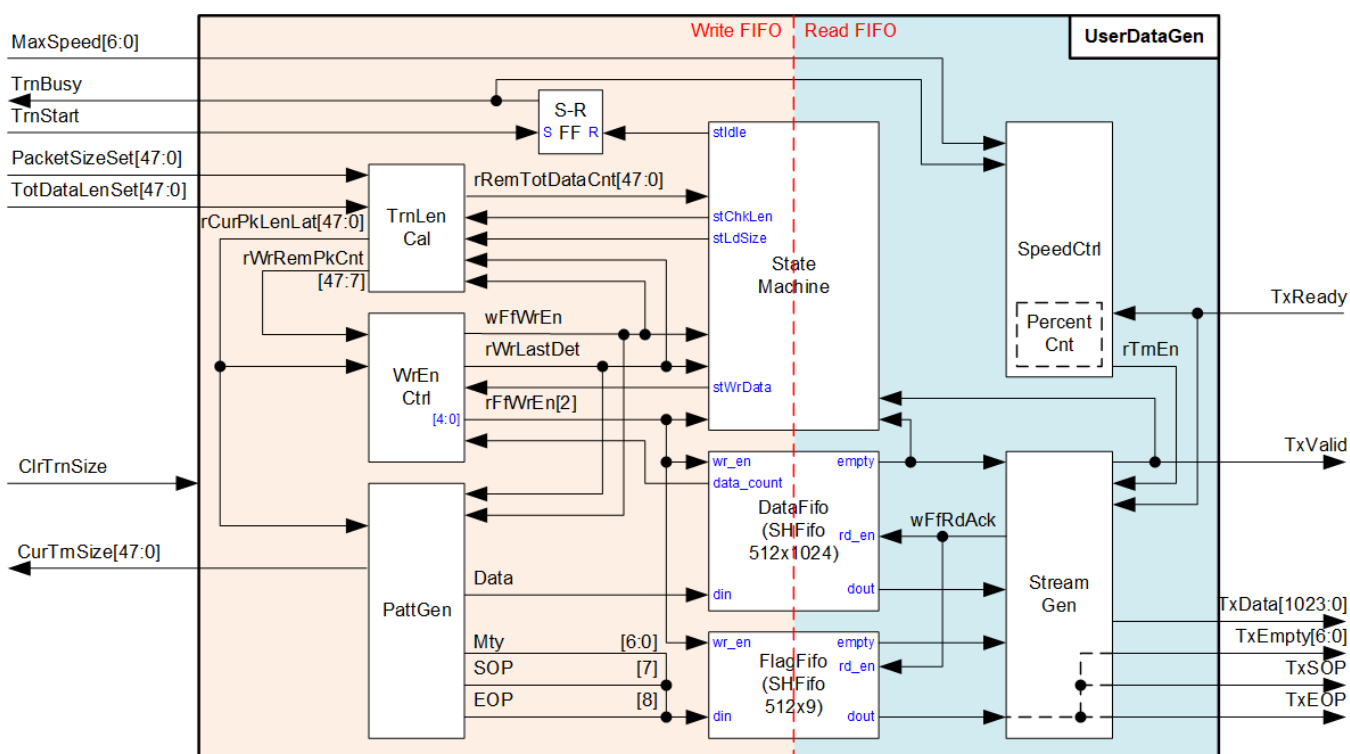


Figure 2-14 UserDataGen block diagram



Figure 2-14 illustrates the block diagram of UserDataGen. The module incorporates two Show-Ahead FIFO named DataFifo and FlagFifo, which serve as buffers for the generated test data and packet flags before streaming them out as Avalon-ST. Two FIFOs are necessary because the data size exceeds the maximum value supported by the IP wizard, which is 1024 bits. The write and read timing for both FIFOs are similar. The logic of UserDataGen is categorized into two groups based on the FIFO – the Write and Read sides.

### FIFO Write Side

A state machine generates flow control signals for writing each data packet to DataFifo and FlagFifo. The operation concludes when the total write data size reaches the TotDataLenSet value. Three key logic functions collaborate with the state machine for the Write function, as detailed below.

- 1) TrnLenCal (Transfer Length Calculation): A group of counters providing information about the necessary transfer size.
  - rRemTotDataCnt: Tracks the remaining transfer size in bytes, initialized from TotDataLenSet and decreasing after each packet is written to the FIFO.
  - rCurPkLenLat: Indicates the data size of each packet in bytes. Typically, each packet size equals PacketSizeSet, except for the last packet. If TotDataLenSet does not align with PacketSizeSet, the final packet size is loaded from rRemTotDataCnt.
  - rWrRemPkCnt: A down-counter indicating the remaining cycles for writing the current packet to the FIFO, initially set to the value of rCurPkLenLat.
- 2) WrEnCtrl (Write Enable Control): Manages the assertion of FIFO write enable (wFfWrEn and rFfWrEn, sourced by wFfWrEn with specific latency time for signal synchronization) and the last transfer cycle of the packet (rWrDataLast). Data is written to the FIFO when the state machine enters the 'stWrData' state, and the FIFO data counter indicates sufficient free space.
- 3) PattGen (Pattern Generator): Generates 32-bit incremental test data and packet flags, such as Start of packet flag (SOP), End of packet flag (EOP), and Empty Byte (Mty), to write to the FIFOs. The initial value of test data is zero and resets to zero when a new start flag (TrnStart) or ClrTrnSize is asserted. PattGen also manages CurTrnSize for user monitoring.

### FIFO Read Side

The FIFO Read side comprises two logic groups: SpeedCtrl and StreamGen.

- 1) SpeedCtrl: Introduces pause time for transmitting the data stream. This logic uses a counter to assert and de-assert the rTrnEn signal within a 100-transfer cycle period, controlled by the MaxSpeed value. During this period, rTrnEn is set to 1b for MaxSpeed cycles and then set to 0b for the remaining (100 – MaxSpeed) cycles. If MaxSpeed equals 100, rTrnEn is continuously set to 1b.
- 2) StreamGen: Reads data from the FIFO when rTrnEn is asserted. It considers the readiness of the read data (FIFO Empty) and the stream interface (TxReady) to decide when to read from FIFO. Finally, the read data is formatted into the output Avalon stream of UserDataGen.

Further operation details of UserDataGen are depicted as timing diagram presented in Figure 2-15 and Figure 2-16.

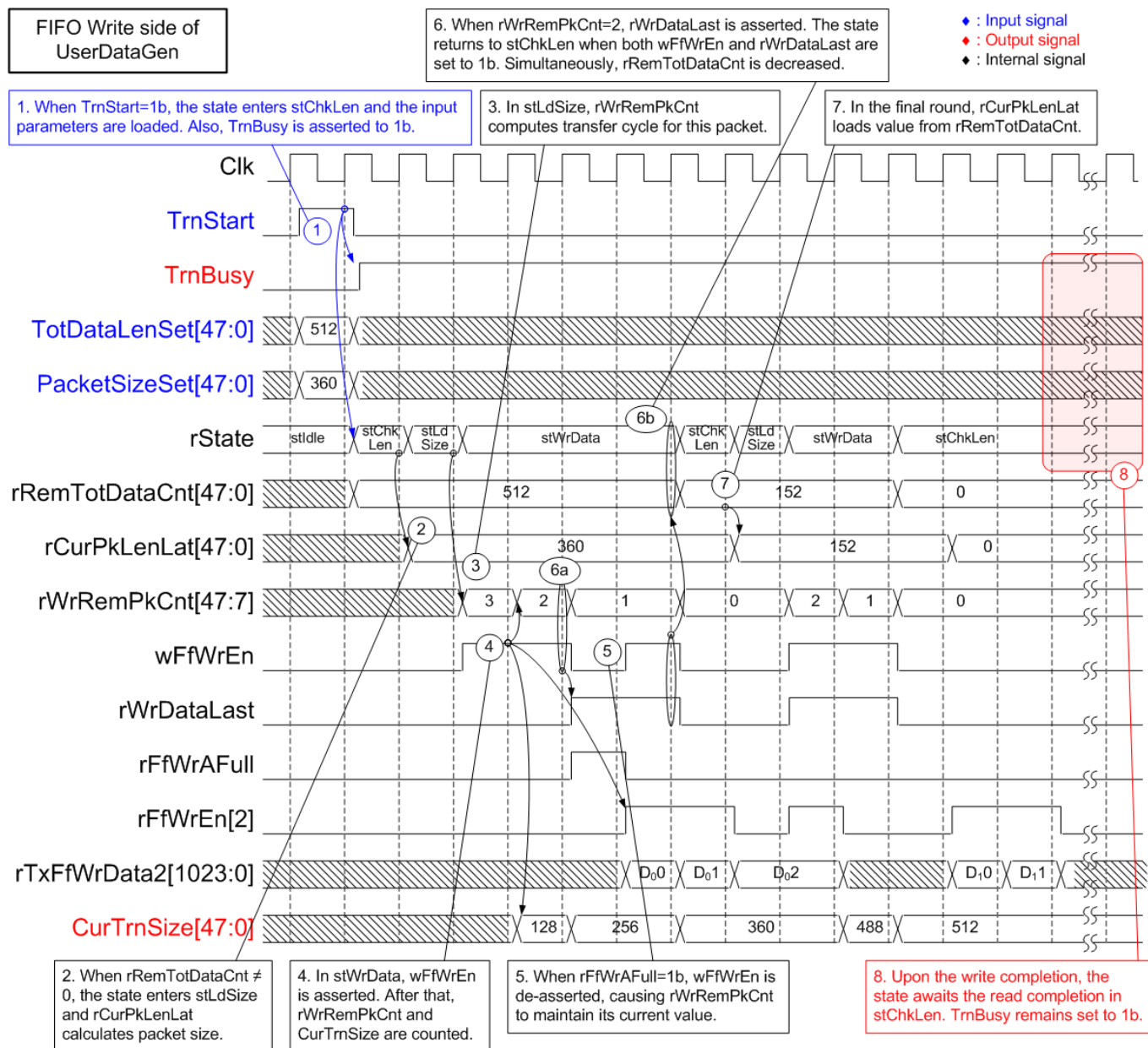


Figure 2-15 Timing diagram of FIFO Write side in UserDataGen

- 1) The operation commences when the user asserts TrnStart to 1b. The state enters stChkLen and TrnBusy is asserted to 1b. At this point, the value of TotDataLenSet and PacketSizeSet are loaded to initialize counters in the TrnLenCal logic, one of which is rRemTotDataCnt.
- 2) In stChkLen, rRemTotDataCnt is examined to determine the next state. If rRemTotDataCnt is non-zero, the state machine enters stLdSize to prepare for packet writing. Additionally, rCurPkLenLat calculates the current packet size, which is typically equal to PacketSizeSet for every packet, except for the last packet.
- 3) 'stLdSize' is one-clock cycle state for reading rCurPkLenLat to compute the rWrRemPkCnt value, a down-counter indicating the remaining cycle amount for writing the current packet to FIFO. Subsequently, the state enters stWrData.
- 4) 'stWrData' is the period for writing one packet data and flags to FIFOs. The WrEnCtrl logic asserts wFfWrEn to write FIFOs if the FIFOs have sufficient free space (rFfAFull=0b). When wFfWrEn is asserted, rWrRemPkCnt is decreased, and FIFO write enable (rFfWrEn[2]) is asserted in the next three clock cycles. Additionally, CurTrnSize is incremented to inform the user about the current transfer size.
- 5) If FIFOs have insufficient free space, rFfWrAFull is set to 1b, resulting in the immediate de-assertion of wFfWrEn. Also, rFfWrEn[2] is de-asserted to 0b in the next three clock cycles.
- 6) When rWrRemPkCnt=2 and wFfWrEn=1b, rWrDataLast is set to 1b, indicating that the next data is the last data of the current packet. After wFfWrEn is asserted for writing the last data, rRemTotDataCnt is decreased by the packet size (rCurPkLenLat), and the state machine returns to stChkLen for checking the next packet writing.
- 7) If the state enters stChkLen for sending the last packet, rCurPkLenLat loads the packet size directly from rRemTotDataCnt, which may be less than PacketSizeSet set by user.
- 8) Once the state returns to stChkLen and rRemTotDataCnt=0, the write operation nearly completed. The last packet is going to be written to the FIFOs in the next few cycles (accounting for D Flip-flops latency). At this point, the state awaits in stChkLen until both the Write and Read operations are completed. TrnBusy is still asserted at 1b.

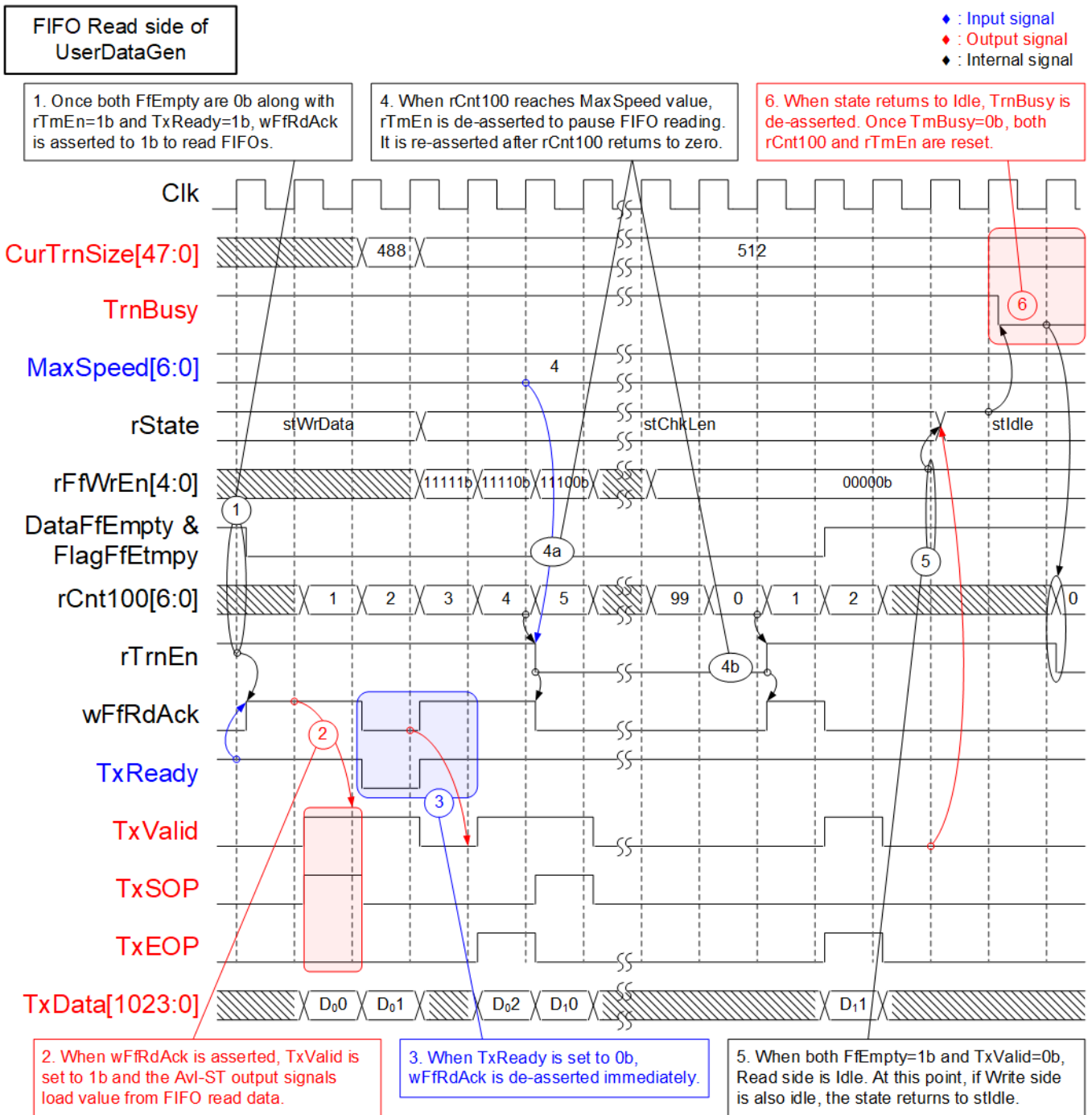


Figure 2-16 Timing diagram of FIFO Read side in UserDataGen

- 1) The read FIFO operation is initiated when three conditions are met: FIFOs must have data (DataFfEmpty=0b and FlagFfEmpty=0b), the transfer speed should not exceed the set value (rTrnEn=1b), and TOE200GADV-IP must be ready to receive data (TxReady=1b). If all three conditions are met, wFfRdAck is asserted to read data out from FIFOs.
- 2) Once wFfRdAck is asserted to 1b, FIFO read data is loaded to become the output signals of the Avalon-stream, with the assertion of TxValid in the next cycle.
- 3) During data transfer, if TxReady is de-asserted to 0b, wFfRdAck is immediately de-asserted to pause data reading.
- 4) Additionally, wFfRdAck can be de-asserted by SpeedCtrl, which includes a counter (rCnt100) to split the transfer cycle in a 100-cycle loop. When rCnt100 reaches MaxSpeed value, rTrnEn is set to 0b to pause FIFO reading. The read operation resumes when rCnt100 resets to zero, leading to the re-assertion of rTrnEn.
- 5) When there is no remaining data in the FIFO (DataFfEmpty=1b and FlagFfEmpty=1b) and no packet transfer in Avalon-ST I/F (TxValid=0b), the read operation is completed. If the write operation is also completed, with rFfWrEn[5:0] equal to zero, the state returns to stIdle.
- 6) In stIdle, TrnBusy is de-asserted to 0b to indicate the completion of the operation. rTrnEn and rCnt100 are also reset from the de-assertion of TrnBusy. Although UserDataGen completes its operation, CurTrnSize maintains its value for user monitoring until a new start (TrnStart) or a clear signal (ClrTrnSize) is set.

### User Data Verification

Similar to UserDataGen, UserDataVer is a submodule within UserReg designed to facilitate high-speed data reception from TOE200GADV-IP. Four UserDataVer modules are specifically allocated for receiving data from four TCP sessions of TOE200GADV-IP. If the user sets the enable flag (RecvEn) to 1b, this module asserts the ready signal to receive the data stream via the Avalon stream and subsequently verifies it. In RecvEn is not asserted, the ready signal is de-asserted, pausing data transmission from TOE200GADV-IP.

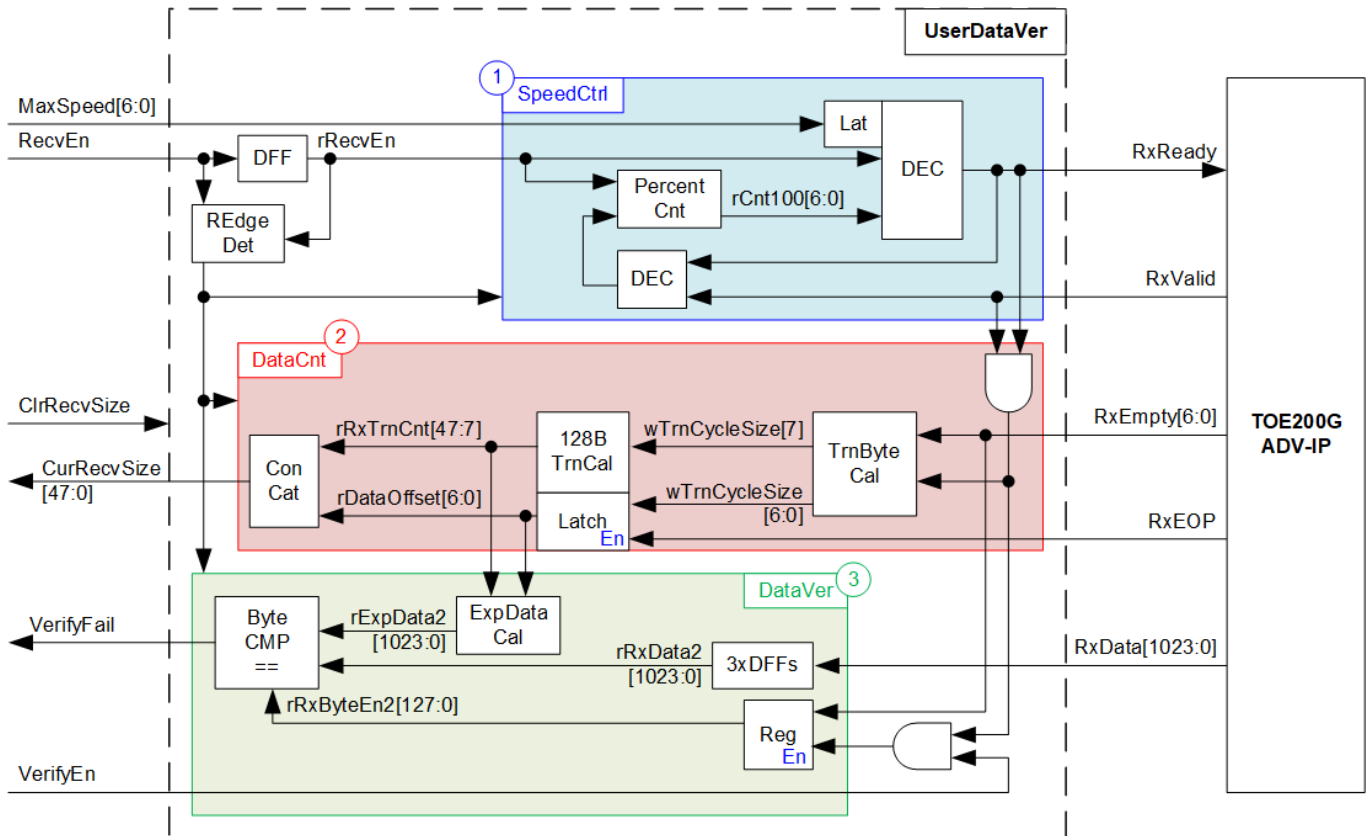


Figure 2-17 UserDataVer block diagram

As shown in Figure 2-17, UserDataVer incorporates three operations:

- 1) Speed Control (Block 1): Controls the transfer speed.
- 2) Data Counter (Block 2): Counts the amount of received data.
- 3) Data Verification (Block 3): Verifies the received data.

Most of these logics initialize their values upon detecting the rising edge of the RecvEn signal, initiating their respective operations. Further details of each operation are described below.

- 1) Speed Control: The logic employs the same concept as the corresponding block in DataGen. An up-counter (rCnt100) counts from 0 to 99, establishing a 100-clock transfer cycle during the operation. RxReady is initialized to 1b when rCnt100 equals zero. Upon rCnt100 reaching MaxSpeed, RxReady is de-asserted to 0b, pausing data transmission from TOE200GADV-IP. RxReady is subsequently re-asserted to 1b when rCnt100 returns to zero. The user-input, MaxSpeed, is loaded upon the new assertion of RecvEn.

- 2) **Data Counter:** This logic calculates and displays the amount of data received from TOE200GADV-IP. Typically, 128-byte data is received in each transfer cycle, indicated when RxEmpty is zero. The last cycle may contain 1-128 bytes of data. The Data Counter comprises two main calculation units. TrnByteCal computes the last valid byte of each received data. Bit[7] of this signal is asserted when the received data reaches a 128-byte unit, while bits[6:0] are loaded as the data offset (rDataOffset) at the end of the current packet, to be used for the next packet transfer. The second calculation unit, 128BTrnCal, increments when bit[7] of wTrnCycleSize is asserted, indicating the amount of received data in 128-byte units, as shown by the rRxTrnCnt signal. The total amount of received data in bytes, which does not align with 128-byte units, is indicated by rDataOffset. Both rRxTrnCnt and rDataOffset can be cleared by either the rising edge of a new RecvEn or by setting the clear flag (ClrRecvSize) to 1b.
  
- 3) **Data Verification:** This logic includes ExpDataCal logic, which calculates the expected data based on the received data counter generated in Block 2. The expected pattern data (ExpData1) is determined as a 32-bit incremental pattern. If RecvEn (receive enable flag) and VerifyEn (verification enable flag) are both activated, each byte of the expected data is compared with the received data stream (rRxData1). In case of a data verification error, the Fail flag (VerifyFail) is set to 1b to notify the user of a mismatch in the data stream. This Fail flag remains set to 1b until the user asserts a new RecvEn or ClrRecvSize.

### 3 CPU Firmware on FPGA

The reference design uses a bare-metal OS for the CPU firmware operating, which facilitates hardware handling. When executing the test system, the first step is to initialize the hardware, described in more details below.

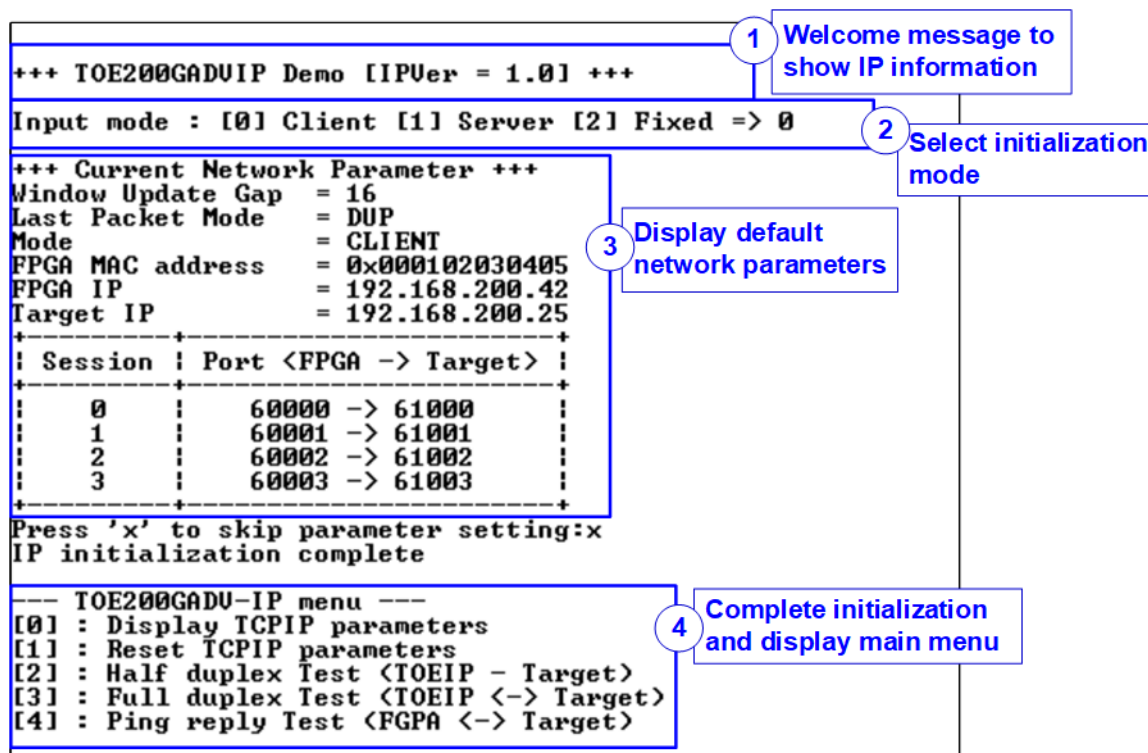


Figure 3-1 System initialization in Client mode by using default parameters

Figure 3-1 illustrates the four-step process for hardware initialization, detailed below.

- 1) Upon FPGA boot-up, the CPU initializes peripherals such as UART and timer. Subsequently, a welcome message is presented on the console via the UART interface. The CPU then awaits Ethernet link establishment, monitored through polling EMAC\_STS\_INTREG[0]. Once the Ethernet link is established, the console displays a menu for parameter configuration.
- 2) Users can choose from three IP initialization options: Client, Server, or Fixed-MAC mode. These modes offer the flexibility to obtain the MAC address of the target device. Client and Server options can be chosen only when the TOE200GADV-IP and the target device are installed in the same network domain. Otherwise, Fixed-MAC mode is required, allowing users to set the MAC address of the target device based on the Ethernet switch connected to TOE200GADV-IP. Details for each initialization mode are outlined as follows.
  - a) *Client mode:* TOE200GADV-IP sends an ARP request packet to retrieve the MAC address of the target device from the ARP reply packet.
  - b) *Server mode:* TOE200GADV-IP waits for an ARP request packet, decodes the MAC address upon reception, and responds with an ARP reply packet.
  - c) *Fixed-MAC mode:* Users manually set the MAC address of the target device to a constant value.

In summary, within a two-FPGA board test environment, users can configure three initialization options: Client<->Server, Client<->Fixed-MAC, and Fixed-MAC<->Fixed-MAC.



- 3) The CPU displays default values for network parameters: the Window update gap value, Last packet mode, initialization mode, FPGA MAC address, FPGA IP address, Target IP address, four FPGA port numbers, and four Target port numbers. The firmware offers two default parameter sets customized for distinct initialization modes: the Server parameter set (used for Server mode only) and Client parameter set (used for both Client and Fixed-MAC modes). In Fixed-MAC mode, an additional parameter, Target MAC address, is also displayed. Users can choose to conclude the initialization process using default parameters or modify specific parameters before initiating the process. The process for updating parameters is outlined in the Reset parameters menu (refer to section 3.2).
- 4) The CPU awaits the completion of IP the initialization process, signaled by TOE\_INF\_INTREG[0] being set to 1b. Upon completion, the console displays the message "IP initialization complete" and the main menu, presenting five options. Detailed explanations for each menu option are provided in the subsequent sections.

### 3.1 Display parameters

This menu serves to present the current parameters of TOE200GADV-IP, i.e., Windows update threshold, last packet mode, initialization mode, source (FPGA) MAC address, source (FPGA) IP address, Target MAC address (displayed exclusively in Fixed MAC mode), Target IP address, all source (FPGA) port numbers, and all Target port numbers. The sequence for displaying parameters is outlined below.

- 1) Read all parameters from each variable within the firmware.
- 2) Print out each variable.

*Note: Source parameters refer to the FPGA parameters set to TOE200GADV-IP, while target parameters are the parameters of a PC or another FPGA.*

### 3.2 Reset parameters

This menu facilitates to modify TOE200GADV-IP parameters, including the initialization mode, IP address, and source port number. As each parameter undergoes an update to be TOE200GADV-IP input signals, the CPU asserts reset signals to both TOE200GADV-IP and user modules. Following the parameter update process, the reset signal is de-asserted, initiating the IP initialization. The sequence for resetting the IP is outlined below.

- 1) Display the latest value of all parameters on the console, similar to section 3.1 (Display parameters).
- 2) Prompt the user to proceed with the current parameters or update their values.
  - a) Press 'x' on the keyboard to skip to step 4 using the latest parameters.
  - b) Press any other key to modify the parameter values, proceeding to step 3.
- 3) Receive input parameters from the user through the following steps.
  - i) Receive the initialization mode from the user. If the initialization mode changes, display the latest parameter set for the new mode on the console. The user can select 'x' to apply the latest parameter set and proceed to step 4. If not, the console requests the next parameter setting (step 3-ii).
  - ii) Receive the remaining parameters from the user and validate each input individually. If an input is deemed invalid, that specific parameter will not be updated.
- 4) Assert a reset to TOE200GADV-IP, UserDataGen, and UserDataVer by setting HW\_RST\_INTREG[0] to 1b.

- 5) Set all parameters of TOE200GADV-IP to the registers such as TOE\_SML\_INTREG and TOE\_DIP\_INTREG.
- 6) De-assert the hardware reset by setting HW\_RST\_INTREG [0] to 0b. Subsequently, TOE200GADV-IP initiates the initialization process.
- 7) Await the completion of the IP initialization process, signified by TOE\_INF\_INTREG[0] being set to 1b.

### 3.3 Half Duplex Test

This menu facilitates the one-way transfer of data for each session. Users can designate the transfer mode for each session independently, choosing among Send test, Receive test, or no operation. Following this, users input parameters such as transfer size, maximum speed, and connection mode (active open for Client mode or passive open for Server mode) to initiate data transmission. It is noted that required parameters vary between “Send test” and “Receive test”, and any invalid inputs automatically cancel the operation.

When “Send test” is selected, the logic generates and transmits 32-bit incremental data to the target device. On the other hand, in a “Receive test”, users can choose to enable or disable data verification while receiving data. If the target device is a PC transmitting dummy data to showcase optimal performance, disabling the data verification function in the logic is recommended to prevent a verification error. The sequence for half-duplex data transfer is outlined below.

- 1) Display the port number of the current session.
- 2) Prompt the user to input transfer mode, transfer size, packet size/data verification mode, maximum transfer speed, and connection mode, and validate all inputs.
- 3) Iterate steps 1) – 2) to execute subsequent sessions until the current session is the final one. The operation is cancelled if all sessions are set to ‘no operation’ transfer mode.
- 4) Configure UserReg registers based on the transfer mode.
  - a) For Send test: Set maximum speed, packet size, and transfer size to the registers (USR0TX\_TRS\_INTREG, USR0TX\_PKLL/H\_INTREG, USR0TX\_LEN/H\_INTREG, respectively).
  - b) For Receive test: Set maximum speed to USR0RX\_TRS\_INTREG.
- 5) Read the connection mode of the current active session. If it is ‘passive mode’, set the current state variable to ST\_CONN\_WAIT. Configure parameters for initiating passive open using user-defined values: FPGA port number (TOE\_SPN\_INTREG) and TCP Last mode (TOE\_LMD\_INTREG). Set the command register (TOE\_CMD\_INTREG[1:0]) to send the passive open request to the specified session. Repeat this step until setup is completed for all active sessions configured by passive open mode.
- 6) Display recommended test application parameters on the PC by reading the current system parameters.
- 7) If there are active sessions configured by active open mode, display message "Press any key to proceed". This message indicates that the user should start port listening (passive open) on the target device by running the test application on the PC or the hardware on the FPGA. Upon completion of passive open execution, users can advance to the next step by entering keys.

- 8) To initiate the active open operation for each session, the current state variable is set to ST\_CONN\_WAIT, and the parameters are configured according to user-defined values: FPGA port number (TOE\_SPN\_INTREG), Target port number (TOE\_DPN\_INTREG), and TCP Last mode (TOE\_LMD\_INTREG). The command register (TOE\_CMD\_INTREG[1:0]) is then set to send the active open request to a specified session. Repeat this step until all active sessions configured by active open mode are set up.
- 9) The operation sequence for transferring data in each active session is as follows, iterated until all active connections are closed. Four state machines are designated to indicate the connection status of each active session: ST\_CONN\_WAIT for waiting connection establishment, ST\_CONN\_ON for transferring data, ST\_CONN\_OFF for no connection, and ST\_CONN\_ERR for error operation.

**ST\_CONN\_WAIT:** This state is designated to wait for connection establishment, monitoring by reading TOE\_CON\_INTREG. Once established, the current state variable changes to ST\_CONN\_ON. Additionally, either UserDataGen or UserDataVer begins operation by setting the command register (USR0TX\_CMD\_INTREG or USR0RX\_CMD\_INTREG). In case of failure in active open command (indicated by TOE\_INT\_INTREG[8] being 1b and TOE\_CCS0\_INTREG[4] being 0b), the current state variable changes to ST\_CONN\_ERR. Once all active sessions complete connection establishment, display the connection information on the console, including the Target port number, MSS value, and Target Window Size. This information is obtained by reading the connection status from TOE200GADV-IP (TOE\_TCS0L/H\_INTREG).

**ST\_CONN\_ON:** This state is designated for transferring data and closing the connection upon the completion of the 'Send test'.

#### Send test

- i) If the connection close command is in process, the state transitions to other states based on two conditions. First, if the connection is terminated successfully, monitored by TOE\_CON\_INTREG, the state enters ST\_CONN\_OFF, and the total number of active sessions is decremented. Second, if command failure is detected (indicated by TOE\_INT\_INTREG[8] being 1b and TOE\_CCS0\_INTREG[4] being 0b), the current state changes to ST\_CONN\_ERR, and the total number of active session is decremented.
- ii) If UserDataGen completes its operation, indicated by USR0TX\_CMD\_INTREG[0]=0b, ensure that all data have been transmitted successfully by reading the amount of remaining data from TOE\_TTS0\_INTREG[15:0]. If all data transmission has been confirmed, request the active close command by setting TOE\_CMD\_INTREG[1:0] with an assertion flag to indicate the connection close command is in process. If the connection is terminated before sending the close request, set the state variable to ST\_CONN\_ERR.
- iii) Ensure that the connection is still active (monitored by TOE\_CON\_INTREG) and update the test progress by retrieving and displaying the total amount of transmitted data (USR0TX\_LEN/L/H\_INTREG) on the console every second. If the connection is inactive before completing data transmission, set the state variable to ST\_CONN\_ERR and decrement the total number of active sessions.

### Receive test

- i) Monitor the connection status by reading TOE\_CON\_INTREG. When it changes to OFF, set the state variable to ST\_CONN\_OFF. While the connection is active, update the test progress by retrieving and displaying the total amount of received data (USR0RX\_LEN/H\_INTREG) on the console every second.
- ii) Upon completion of data transfer, indicated by the connection status changing to OFF, read the verification result (USR0RX\_CMD\_INTREG) and compare the total amount of received data with the set value. If verification failure or a mismatch in the total number of received data is found, display an error message.

10) Calculate performance and display the test result on the console.

## 3.4 Full duplex test

This menu enables bidirectional data transfer for each session using a single target device, which can be a PC or FPGA. Users can independently enable data transfer of each session, choosing between full-duplex testing or no operation. Following this, users input parameters such as total transfer size and connection mode (active open/close for Client mode or passive open/close for Server mode). The transfer size set by the user must match the size set on the target device. If utilizing the test application 'tcp\_client\_txrx\_single' on PC for testing, the connection mode on the FPGA must be set to Server mode to execute passive open and close. The sequence for full-duplex data transfer is outlined below.

- 1) Receive parameter inputs from users, following a process similar to the steps 1) through 3) of the Half Duplex Test.
- 2) If the transfer mode is enabled, configure UserReg registers accordingly. Set maximum speed for both transfer directions, packet size, and transfer size to the registers (USR0TX/RX\_TRS\_INTREG, USR0TX\_PKLL/H\_INTREG, USR0TX\_PKLL/H\_INTREG, respectively).
- 3) Initiate the opening of a connection based on the specified connection mode, similar to the process outlined in the steps 5 through 7 of the Half Duplex Test.
- 4) The operation sequence for transferring data in each active session is as follows, iterated until all active connections are closed. Four state machines are designated, similar to those in the Half Duplex Test; however, the operation in each state differs slightly different from that in the Half Duplex Test, as described below.

**ST\_CONN\_WAIT:** The operation in this state is similar to that of the Half Duplex Test, with the distinction that both UserDataGen and UserDataVer begin operation by setting the command register (USR0TX/RX\_CMD\_INTREG) for bidirectional data transfer.

**ST\_CONN\_ON:** This state is designated for bidirectional data transfer. Upon completion of all data transfer, the connection close is requested in this state, configured by active open/close mode. The completion process for each connection mode (active or passive) slightly differs, so the execution details for each mode are split into two parts, as follows.

Passive mode

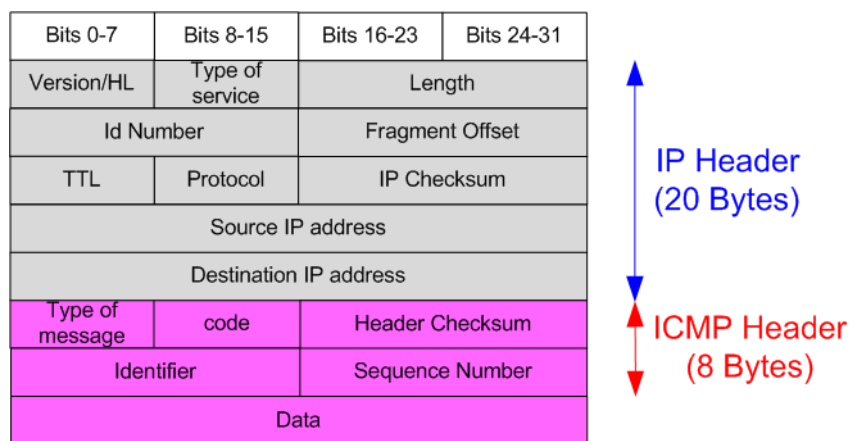
- i) Monitor the connection status by reading TOE\_CON\_INTREG. If it changes to OFF, proceed to the next step. Otherwise, update the test progress by retrieving and displaying the total amount of transmitted data and received data on the console every second, reading from USR0TX/RX\_LEN/H\_INTREG.
- ii) Ensure that all data is transmitted successfully, indicated by the de-assertion of UserDataGen's busy flag (USR0TX\_CMD\_INTREG[0]=0b) and the remaining transmitted data amount equal to zero (TOE\_TTS0\_INTREG[15:0]). Upon the successful completion of data transfer, set the state variable to ST\_CONN\_OFF. Otherwise, set the state variable to ST\_CONN\_ERR. During transitioning to other states, decrement the total number of active sessions.

Active mode

- i) If the connection close command is in process, the state transitions to other states based on two conditions. First, if the connection is terminated successfully, monitored by TOE\_CON\_INTREG, the state enters ST\_CONN\_OFF, and the total number of active sessions is decremented. Second, if command failure is detected (indicated by TOE\_INT\_INTREG[8] being 1b and TOE\_CCS0\_INTREG[4] being 0b), the current state changes to ST\_CONN\_ERR, and the total number of active session is decremented.
  - ii) If the connection is terminated before UserDataGen completes its operation, set the state variable to ST\_CONN\_ERR and decrement the total number of active sessions. Otherwise, proceed to the next step.
  - iii) Monitor the completion status of UserDataGen, indicated by USR0TX\_CMD\_INTREG[0] set to 0b and TOE\_TTS0\_INTREG[15:0] equal to 0. If it does not complete, update the test progress by retrieving and displaying the total amount of transmitted data and received data on the console every second, reading from USR0TX/RX\_LEN/H\_INTREG. If the UserDataGen operation is completed, it needs to wait for the completion of UserDataVer, indicated by USR0TX\_LEN/H\_INTREG matching the transfer size set by the user. Once all data transfer is successfully completed, request the active close command by setting TOE\_CMD\_INTREG[1:0] with an assertion flag to indicate the connection close command is in process.
- 5) Read the verification result (USR0RX\_CMD\_INTREG) and compare the total amount of received data with the set value. If verification failure or a mismatch in the total number of received data is found, display an error message.
  - 6) Calculate performance and display the test result on the console.

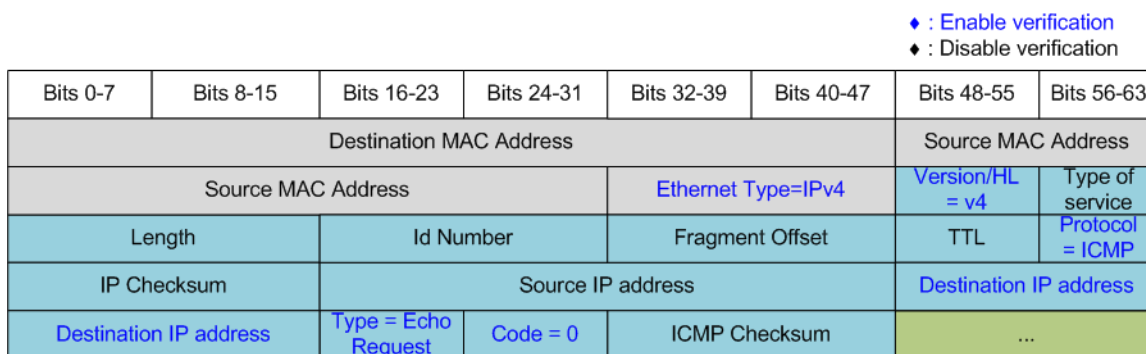
### 3.5 Ping reply test

When the PC runs the Ping command to check round-trip time, it generates an ICMP Echo request packet. This menu is designed to configure the hardware to receive ICMP Echo request packets. Upon receiving a valid request packet, the hardware generates an ICMP Echo reply as the response packet. The packet structure of the 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 8, while the 'Type' value of the Echo reply is 0. For more information about the Ping command, please refer to the following website. [http://en.wikipedia.org/wiki/Ping\\_\(networking\\_utility\)](http://en.wikipedia.org/wiki/Ping_(networking_utility))



**Figure 3-3 ICMP Echo request packet filtering**

The sequence to run the Ping reply is described as follows.

1) Call the 'init\_filter' function to set the filtering parameters of UserRxMAC, allowing it to receive only ICMP Echo request packets. The configuration values for the ICMP Echo request packets are as follows (highlighted in blue 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 the FPGA
- ICMP type (1 byte) = 0x08 (Echo Request)
- ICMP code (1 byte) = 0x00 (Echo Request code)

*Note:* Figure 3-3 shows only a 38-byte ICMP packet. The actual size of the ICMP packet is 42 bytes, excluding the reset of the header field, because the filtering logic in the UserRxMAC module is designed to support up to 38-byte header data.

- 2) Enable the UserRxMAC module by setting RXEMAC\_CMD\_INTREG[0] to 1b.
- 3) Wait until a new packet is stored in RxRAM, indicated by the empty flag of RxMacFf (RXEMAC\_FF\_INTREG[15]) being equal to 0b.
- 4) Read and validate the last address of the received packet from RxMacFf (RXEMAC\_FF\_INTREG[4:0]). After that, assert read acknowledge to flush the current read data from RxMacFf (RXEMAC\_CMD\_INTREG[1]=1b).
- 5) Copy the received packet from RxRAM (RXRAM\_BASE\_ADDR) to the receive temporal buffer (rxbuff\_ch).
- 6) Decode the received packet and proceed to the next step if the packet is an Echo request packet and the parameters, including checksum, are correct. Otherwise, display an error message.
- 7) Prepare the Echo reply packet in the transmit temporal buffer (txbuff\_ch), including the calculated IP checksum and ICMP checksum. After that, copy data from the transmit temporal buffer (txbuff\_ch) to TxRAM (TXRAM\_BASE\_ADDR).
- 8) Set UserTxEMAC register to start data sending by setting TXMAC\_LEN\_INTREG to the length of the Echo reply packet.
- 9) Return to step 2 to proceed with the next packet processing. This forever loop can be interrupted when users press any key on the console. To complete the operation, all data from both RxMacFf and RxRAM is removed, and the UserRxMAC module is disabled by setting RXEMAC\_CMD\_INTREG[0]=0b to halt low-speed port processing. Finally, return to the main menu.

### 3.6 Function list in CPU firmware

This section outlines the function list, categorized into two groups: functions for executing high-speed connection and low-speed connection. Further details for each function are described as follows.

#### 3.6.1 Function for High-Speed Connection

unsigned int cal_strlen(unsigned int num)	
Parameters	num: Integer input to calculate the string length
Return value	ret: The length of the string for displaying
Description	Calculate the length of the string required to display this value in integer style, and return the result as the function's return value.

void check_cmd_cpl(unsigned int session, unsigned int* status)	
Parameters	session: The session number status: Returned value to indicate the completion status of command. 0: Processing, 1: Failure, 2: Success.
Return value	None
Description	Decode the read value from the TOE_INT_INTREG register to track the command completion of a specific session, defined by the 'session' parameter. If the completion flag is detected, set TOE_CCS0_INTREG to clear the completion flag, and decode the completion status as success or failure. Finally, return the current status to the 'status' parameter.

void check_conon(unsigned int session, unsigned int* status)	
Parameters	session: The session number status: Returned value to indicate the connection status. 0: Connection OFF, 1: Connection ON.
Return value	None
Description	Read the value from the TOE_CON_INTREG register for the specific session defined by the 'session' parameter, and return the result to the 'status' parameter.

void check_ethlink(unsigned int* status)	
Parameters	status: Returned value to indicate the Ethernet status. 0: Ethernet link down, 1: Ethernet link up.
Return value	None
Description	Read Ethernet MAC link status from EMAC_STS_INTREG, and return the result to the 'status' parameter.

void deactivate_session(unsigned int* wait_conn_off, unsigned int* num_act_session)	
Parameters	wait_conn_off: The flag to indicate that the active close is processing num_act_session: The number of active sessions
Return value	None
Description	Reset the 'wait_conn_off' parameter to indicate the completion of the connection close request and decrement the value of 'num_act_session' parameter. This function is called to clean up the parameters after the connection is terminated.



void init_conn(TEST_PARAM* testparam, unsigned int* num_act_sesssion, unsigned int* num_aop_session, unsigned int* cur_state)	
Parameters	testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed num_act_session: The number of active sessions num_aop_session: The number of sessions configured by active open cur_state: The current state machine value indicating the connection status
Return value	None
Description	This function sets up UserDataGen, UserDataVer, and connection establishment before proceeding with the data transfer during both the Half Duplex Test and the Full Duplex Test.

void init_param(void)	
Parameters	None
Return value	None
Description	Execute the 'Reset parameters' menu according to the description in section 3.2. This involves calling the 'show_param' and 'input_param' functions to display and retrieve parameters from the user, respectively.

int input_param(void)	
Parameters	None
Return value	0: Valid input, -1: Invalid input
Description	Receive network parameters from the user, including the initialization mode, the last packet mode, the window update threshold, FPGA MAC address, FPGA IP address, FPGA port number, Target IP address, Target port number, and Target MAC address (only when using Fixed MAC mode). Each input is validated separately. If a parameter is valid, it will be updated; otherwise, it will remain unchanged. After receiving all parameters, call the 'show_param' function to display them.

int input_test_param(unsigned int test_menu, TEST_PARAM* testparam, unsigned int* num_act_sesssion, unsigned int* num_aop_session)	
Parameters	test_menu: The test menu for execution, which can be half-duplex or full-duplex testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed num_act_session: The number of active sessions num_aop_session: The number of sessions configured by active open
Return value	0: Valid input, -1: Invalid input
Description	Receive test parameters from the user, including operation mode, transfer size, packet size, data verification mode, maximum speed, and connection mode. If any input is invalid, the operation will be cancelled. Upon completion of parameter validation, calculate the number of active sessions and the number of sessions configured by active open mode, and return the results to the 'num_act_session' and 'num_aop_session' parameters, respectively.

void show_cursize(TEST_PARAM* testparam, unsigned int* cur_state, unsigned long long* cur_txsize, unsigned long long* cur_rxsize)	
Parameters	testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed cur_state: The current state machine value indicating the connection status cur_txsize: The current amount of transmitted data, measured in bytes cur_rxsize: The current amount of received data, measured in bytes
Return value	None
Description	Read and display the connection status of all sessions. If the data is transferring, read the current amount of transmitted and received data from USRTX/RX_LEN/H_INTREG, and then display it on the console in Byte, Kbyte, or Mbyte units.

void show_eth_status(void)	
Parameters	None
Return value	None
Description	Read the current status of the Ethernet MAC from EMAC_STS_INTREG, decode it, and display the status on the console.

void show_param(void)	
Parameters	None
Return value	None
Description	Execute the 'Display parameters' menu according to the description in section 3.1.

void show_perf_header(unsigned int *test_mode)	
Parameters	test_mode: The operation mode which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3)
Return value	None
Description	When 'test_mode' parameter is not set to 'No test', display session number as the header of the current status table.

void show_perf_line(unsigned int *test_mode)	
Parameters	test_mode: The operation mode which can be No test (0), Send test (1), Receive test (2), or Full-duplex (3)
Return value	None
Description	When 'test_mode' parameter is not set to 'No test', display a straight line to be a part of the current status table.

void show_port_info(TEST_PARAM* testparam, unsigned int *cur_state)	
Parameters	testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed cur_state: The current state machine value indicating the connection status
Return value	None
Description	Scan the active session by reading the 'testparam' parameter. If the connection status is ON, display its information, including the Target port number, MSS value, and the window scaling factor of the Target by reading TOE_TCS0L/H_INTREG.

void show_reset_int_status(unsigned int session)	
Parameters	session: The session number
Return value	None
Description	Read the Reset interrupt status of a specific session, defined by the 'session' parameter, from TOE_RSS0_INTREG, decode it, and display the message.

void show_result(TEST_PARAM* testparam, unsigned int* cur_state, unsigned long long* cur_txsize, unsigned long long* cur_rxsize, unsigned int *recv_size_err, unsigned int *recv_ver_err)	
Parameters	testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed cur_state: The current state machine value indicating the connection status cur_txsize: The current amount of transmitted data, measured in bytes cur_rxsize: The current amount of received data, measured in bytes recv_size_err: An error flag asserted by the mismatch in receive size recv_ver_err: An error flag asserted by the mismatch in received data
Return value	None
Description	Check error conditions, including the connection error, the receive size error, and the received data error, by reading the 'cur_state', 'recv_size_err', and 'recv_ver_err' parameters, respectively. Display the corresponding error message for each error flag. After that, read USRTX/RX_LEN/L/H_INTREG to display the total amount of transmitted data and received data, respectively. Also, read the timer value to calculate the total time usage for processing and display the result in usec, msec, or sec. Finally, calculate and display transfer performance in MB/s.

void show_retry_int_status(void)	
Parameters	None
Return value	None
Description	Display Retry interrupt status of session which detects Retry interrupt every 4 times from read value of TOE_RSS_INTREG.

void show_space_line(unsigned int *test_mode)	
Parameters	test_mode: Pointer of array that stores test_mode which can be set to No test (0), Send test (1), Receive test (2), or Full-duplex (3)
Return value	None
Description	When test mode is not 'No test' (0), display blank space line to be a part of the current status table.

int toe_full_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Execute the 'Full duplex test' menu according to the description in section 3.4.

int toe_half_test(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Execute the 'Half duplex test' menu according to the description in section 3.3.

void update_cursize(unsigned long long* cur_txsize, unsigned long long* cur_rxsize)	
Parameters	cur_txsize: The current amount of transmitted data, measured in bytes cur_rxsize: The current amount of received data, measured in bytes
Return value	None
Description	Read USRTX_LEN/H_INTREG and USRRX_LEN/H_INTREG to update the current amount of transmitted data and received data to 'cur_txsize' and 'cur_rxsize' parameters, respectively.

void update_curstat(unsigned int* disp_1st_stat, TEST_PARAM* testparam, unsigned int* cur_state, unsigned long long* cur_txsize, unsigned long long* cur_rxsize, unsigned long long* prv_txsize, unsigned long long* prv_rxsize)	
Parameters	disp_1st_stat: The flag to indicate the first line for displaying to show the connection status Testparam: A set of test parameters input from the user, such as operation mode, connection mode, and maximum speed cur_state: The current state machine value indicating the connection status cur_txsize: The current amount of transmitted data, measured in bytes cur_rxsize: The current amount of received data, measured in bytes prv_txsize: The latest value of cur_txsize before updating prv_rxsize: The latest value of cur_rxsize before updating
Return value	None
Description	Check the current state machine and the current transfer size of both directions, then displays all information in table format by calling the 'show_port_info' and 'show_cursize' functions. If additional data is transferred, the interrupt count will be cleared. Finally, check if there is an interrupt asserted in the system. If yes, call 'show_retry_int_status' function to decode and display interrupt status.

void wait_ethlink(void)	
Parameters	None
Return value	None
Description	Read EMAC_STS_INTREG[0] to monitor the Ethernet link status. The function is completed when the Ethernet connection is established.

### 3.6.2 Functions for Low-Speed Connection

unsigned int cal_checksum (unsigned int byte_len, unsigned char *buf)	
Parameters	byte_len: The data length in bytes 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. The user must prepare the array of data in character data type and determine the data length. Then call the function using data length and the character pointer to the first data in the array. The return value is the calculated 16-bit checksum value of the data.

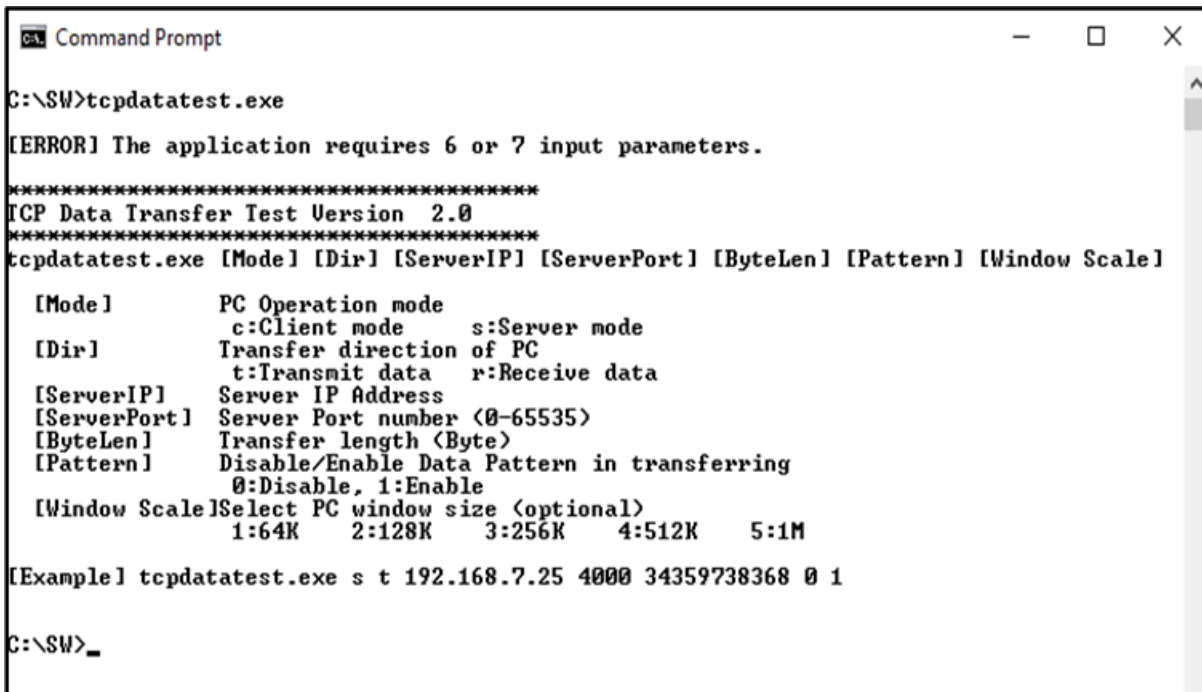
void init_filter (void)	
Parameters	None
Return value	None
Description	Disable UserRxEMAC by setting RXEMAC_CMD_INTREG to block the received packet. Next, Write RXEMAC_HDVAL_ADDR and RXEMAC_HDEN_ADDR for filtering an ICMP Echo request packet, as shown in the blue text of Figure 3-3.

void ping_test (void)	
Parameters	None
Return value	None
Description	Execute the 'Ping reply test' menu according to the description in section 3.5.

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 if there are any received data packets in the low-speed connection. If a new received packet is detected, read the packet length from RxMacFf by reading RXEMAC_FF_INTREG, and flush this read data from RxMacFf by writing RXEMAC_CMD_INTREG[1]=1b. Next, copy data from RxRAM inside UserRxEMAC (RXRAM_BASE_ADDR) to the character array variable. Finally, return the read length in byte units.

## 4 Test Software on PC

### 4.1 'tcpdatatest' application (Half duplex test)



```

C:\SW>tcpdatatest.exe

[ERROR] The application requires 6 or 7 input parameters.

*****
TCP Data Transfer Test Version 2.0
*****
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] Select PC window size (optional)
                 1:64K   2:128K   3:256K   4:512K   5:1M

[Example] tcpdatatest.exe s t 192.168.7.25 4000 34359738368 0 1

C:\SW>_

```

Figure 4-1 'tcpdatatest' application usage

The 'tcpdatatest' application is executed to send or receive TCP payload data on a PC. It requires six mandatory parameters and one optional parameter. It is important to ensure that the parameter inputs match those set on the FPGA. The details of each parameter are as follows.

#### Mandatory parameters

- 1) Mode : c – The PC runs in Client mode and the FPGA runs in Server mode  
s – The PC runs in Server mode and the FPGA runs in Client mode
- 2) Dir : t – transmit mode (the PC sends data to the FPGA)  
r – receive mode (the PC receives data from the FPGA)
- 3) ServerIP : The IP address of the FPGA when the PC runs in Client mode  
(Default is 192.168.200.42)
- 4) ServerPort : The port number of the FPGA when the PC runs in Client mode  
(Default is 60000)
- 5) ByteLen : The total size of data to be transferred in bytes. This parameter is used only in transmit mode only and is ignored in receive mode. In transmit mode, the ByteLen value must match the total transfer size set in the receive data test menu of the FPGA. In receive mode, the application is closed when the connection is terminated.
- 6) Pattern : 0 – Generate dummy data in transmit mode and disable data verification in receive mode.  
1 – Generate incremental data in transmit mode and enable data verification in receive mode.

### Optional parameter

- 1) **Window Scale** : Indicate the size of the allocated buffer for the TCP socket on the PC. It is also applied for TCP Window scaling feature. The valid range is 1-5.
  - 1 – Allocated buffer size of 64 KB
  - 2 – Allocated buffer size of 128 KB
  - 3 – Allocated buffer size of 256 KB
  - 4 – Allocated buffer size of 512 KB
  - 5 – Allocated buffer size of 1 MB

*Note: Window Scale parameter is optional. If the user does not provide this parameter, it is automatically set to 1.*

The sequence of the test application when running in transmit mode and receive mode are described as follows.

### Transmit mode

- 1) Obtain and verify the user's input parameters, excluding the optional one.
- 2) Create a socket, set the socket options, and specify the socket memory size.
- 3) Establish a new connection using the Server IP address and Server port number.
- 4) Allocate 2 MB memory for the send buffer.
- 5) Generate the incremental test pattern to the send buffer if the dummy pattern is not selected.
- 6) Send data out and read the total amount of sent data through the socket function.
- 7) Calculate the remaining transfer size.
- 8) Print the total transfer size every second.
- 9) Repeat step 5) – 8) until the remaining transfer size is 0.
- 10) Calculate the total performance and print the result on the console.
- 11) Close the socket and free the memory.

### Receive mode

- 1) Follow steps 1) – 3) of the Transmit data mode.
- 2) Allocate 2 MB memory for the receive buffer.
- 3) Wait for the data to be stored in the receive buffer and read it, and increase the total amount of received data.
- 4) Verify the received data using the incremental pattern if data verification is enabled. Otherwise, skip this step. Print an error message if the data is incorrect.
- 5) Print the total amount of received data every second.
- 6) Repeat steps 3) – 5) until the connection is closed by the other device.
- 7) Calculate the total performance and print the result on the console.
- 8) Close the socket and free the memory.

## 4.2 'tcp\_client\_txrx\_single' application (Full duplex test)

```

Command Prompt

C:\SW>tcp_client_txrx_single

[ERROR] The application requires 4 or 5 input parameters.

*****
TCP Tx Rx Test Version 1.2
*****
tcp_client_txrx_single [ServerIP] [ServerPort] [ByteLen] [Pattern] [Window Scale]

[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] Select PC window size (optional)
                1:64K   2:128K   3:256K   4:512K   5:1M

[Example] tcp_client_txrx_single 192.168.40.42 60000 137438953440 1 1

C:\SW>

```

Figure 4-2 'tcp\_client\_txrx\_xg' application usage

The 'tcp\_client\_txrx\_single' application allows the PC to send and receive TCP payload data through Ethernet using the same port number simultaneously. It operates exclusively in Client mode and necessitates the input of Server parameters (network parameters of the FPGA) by the user. The application employs five parameters, outlined as follows.

### Mandatory parameters

- 1) ServerIP : The IP address of the FPGA
- 2) ServerPort : The port number of the FPGA
- 3) ByteLen : The total transfer size in byte units, which is the total amount of data for both transfer directions. This value must match the transfer size set on the FPGA for running a full-duplex test.
- 4) Pattern : 0 – Generate dummy data for the sending function and disable data verification for the receiving function. This mode is used to assess the optimal performance of full-duplex transfer.  
1 – Generate incremental data for the sending function and enable data verification for the receiving function.



### Optional parameter

- 1) **Window Scale** : Indicate the size of the allocated buffer for the TCP socket on the PC. It is also applied for TCP Window scaling feature. The valid range is 1-5.
- 1 – Allocated buffer size of 64 KB
  - 2 – Allocated buffer size of 128 KB
  - 3 – Allocated buffer size of 256 KB
  - 4 – Allocated buffer size of 512 KB
  - 5 – Allocated buffer size of 1 MB

*Note: Window Scale parameter is optional. If the user does not provide this parameter, it is automatically set to 1.*

The sequence of the test application is outlined below.

- 1) Obtain and verify the user's input parameters, excluding the optional one.
- 2) Allocate 2 MB memory for the send and receive buffers separately.
- 3) Create the socket, set socket options, and specify the socket memory size.
- 4) Establish a new connection using the Server IP address and Server port number.
- 5) If the test pattern is enabled, generate the incremental test pattern in the send buffer; otherwise, proceed with dummy data.
- 6) If the send function is not ready for operating, skip this step; otherwise, proceed with the following sub-steps.
  - i) If the test pattern is enabled, generate the incremental test pattern in the send buffer; otherwise, skip this step for dummy data.
  - ii) Send data out and read the amount of sent data through the socket function.
  - iii) Calculate the remaining send size.
- 7) If the receive function is not ready to operate, skip this step; otherwise, proceed with the following sub-steps.
  - i) Read data from the receive buffer and increase the total amount of received data.
  - ii) If the test pattern is enabled, verify the received data using the incremental pattern, and print an error message if verification fails; otherwise, skip this step.
- 8) Print the total amount of transmitted data and received data every second.
- 9) Repeat steps 5) – 8) until the total amount of transmitted and received data equals ByteLen, as set by the user.
- 10) Calculate the performance and print the result on the console.
- 11) Close the socket.

## 5 Revision History

Revision	Date	Description
1.00	13-Jun-24	Initial version release