



TOE100GADV-IP reference design

Rev1.00 9-Feb-24

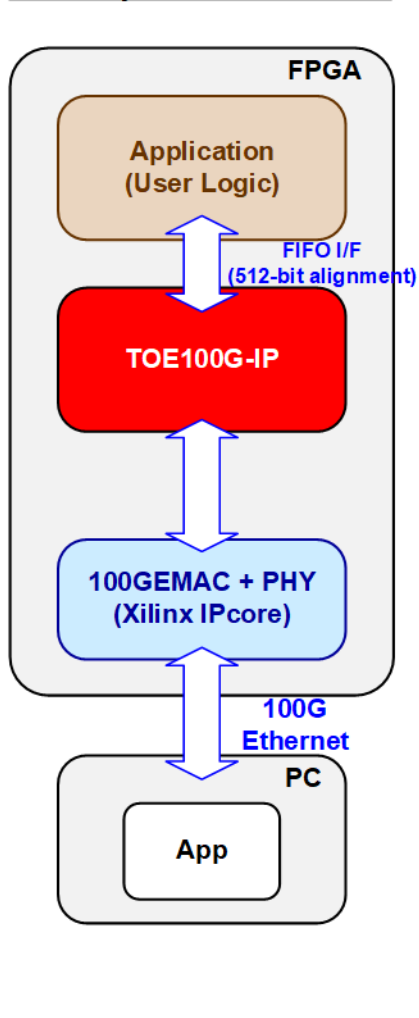
- 1 Introduction2
- 2 Hardware overview5
 - 2.1 100G Ethernet (MAC) Subsystem (100G BASE-SR)7
 - 2.1.1 MACTxAsync512IF8
 - 2.1.2 MACRxAsync512IF11
 - 2.1.3 MAC100GTxIF13
 - 2.1.4 MAC100GRxIF15
 - 2.2 AxiSSw2to118
 - 2.3 TOE100GADV-IP20
 - 2.4 User2MAC21
 - 2.4.1 UserTxMAC22
 - 2.4.2 UserRxMAC24
 - 2.5 CPU and Peripherals26
 - 2.5.1 AsyncAxiReg27
 - 2.5.2 UserReg29
- 3 CPU firmware and Test software42
 - 3.1 Display parameters43
 - 3.2 Reset parameters43
 - 3.3 Half Duplex Test44
 - 3.4 Full duplex test46
 - 3.5 Ping reply test48
 - 3.6 Function list in CPU firmware50
 - 3.6.1 Functions for High-Speed Connection50
 - 3.6.2 Functions for Low-Speed Connection55
- 4 Test Software on PC56
 - 4.1 'tcpdatatest' application56
 - 4.2 'tcp_client_txrx_single' application58
- 5 Revision History60

1 Introduction

Design Gateway has been proposing various networking solutions for a decade, particularly focusing on Gigabit Ethernet communication. One such solution is TOE100G-IP core which implements the Transport and Internet layers of the TCP/IP Protocol using complete hardwired logic. This IP Core showcases a fundamental reference design demonstrating TCP/IP offloading engine functionality for a single TCP/IP communication session, utilizing a single TOE100G-IP core.

However, the performance results of this design often show limitations, when the test environment involves an FPGA and a PC for a single TCP/IP session. In such scenarios, the achieved test performance cannot reach even half of the maximum line speed. However, when using two TOE100G-IPs transferring data between each other, the maximum line speed becomes achievable.

Default system of TOE100G-IP



Multiple-session system of TOE100GADV-IP

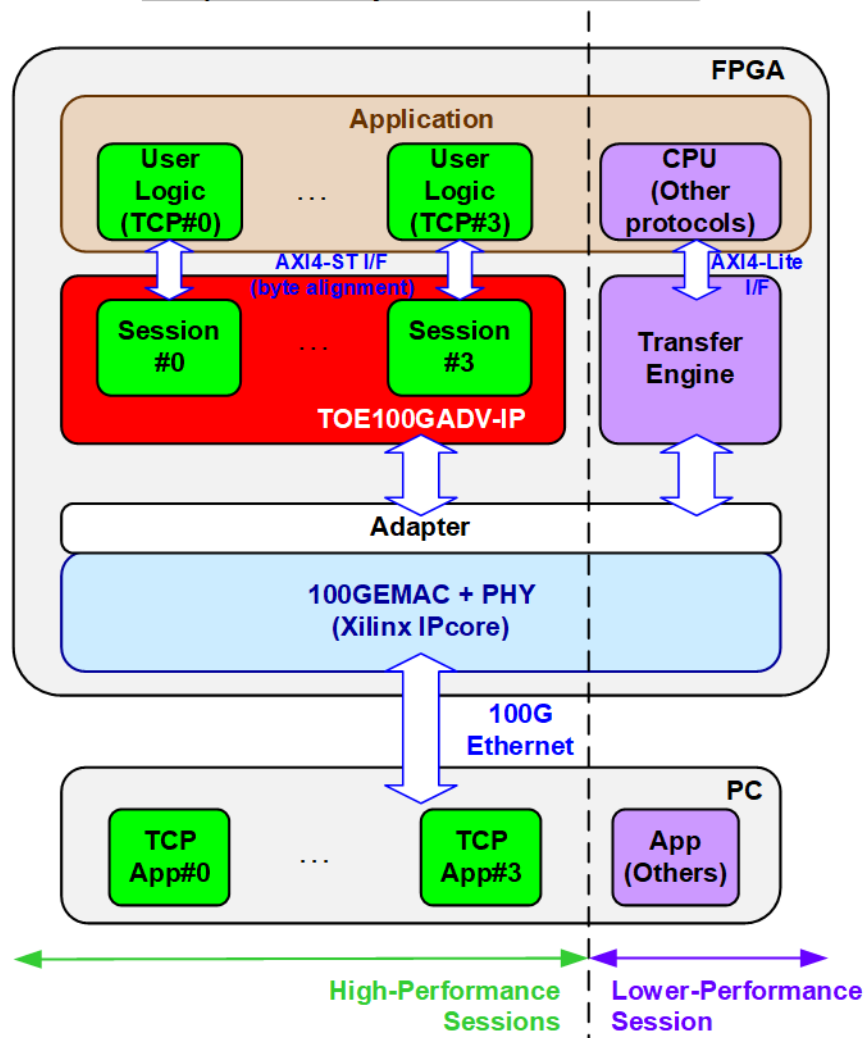


Figure 1-1 System examples between Single-session and Multiple-session solutions

To overcome this constraint, a new IP Core model, the TOE100GADV-IP, has been introduced, featuring a native multi-session architecture. This advanced IP Core is capable of accommodating up to four simultaneous sessions, utilizing the same 100G Ethernet channel and sharing its bandwidth. Leveraging this multiple-session transferring capability notably enhances the overall performance between the FPGA and PC, approaching the theoretical line rate of 100G Ethernet speed. Additionally, this multiple-session IP Core adds versatility and complexity to the system, enabling multi-rate adoption within the application system, as illustrated on the right side of Figure 1-1.

Furthermore, the user interface of the TOE100GADV-IP is compatible with standard Stream I/F such as AXI4-ST I/F, facilitating seamless integration of the IP with other modules through standard interfaces. While the TOE100G-IP employs a 512-bit FIFO interface, the most straightforward interface for user data transfer, it necessitates the size of each transmitted packet to align with 512 bits.

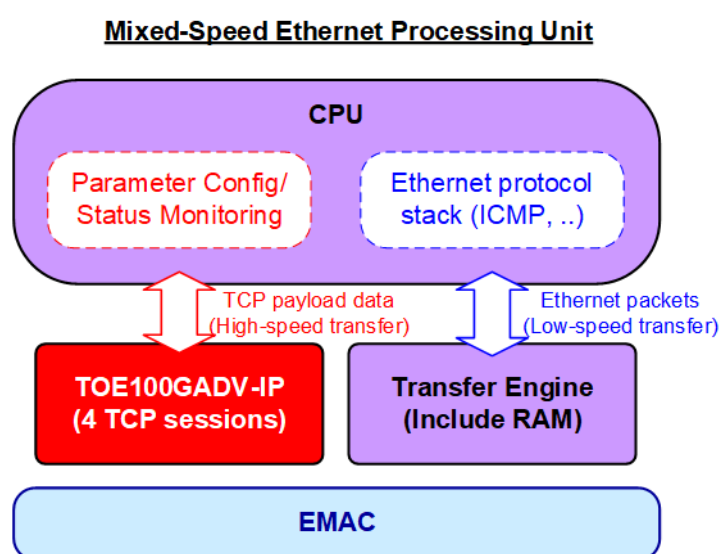


Figure 1-2 Mixed-speed and versatile Ethernet processing unit

The TOE100GADV-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.

The system illustrated in Figure 1-2 serves this purpose effectively. One TOE100GADV-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 TOE100GADV-IP is integrated to process four high-speed TCP payload data. Although the reference design activates all four sessions of TOE100GADV-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

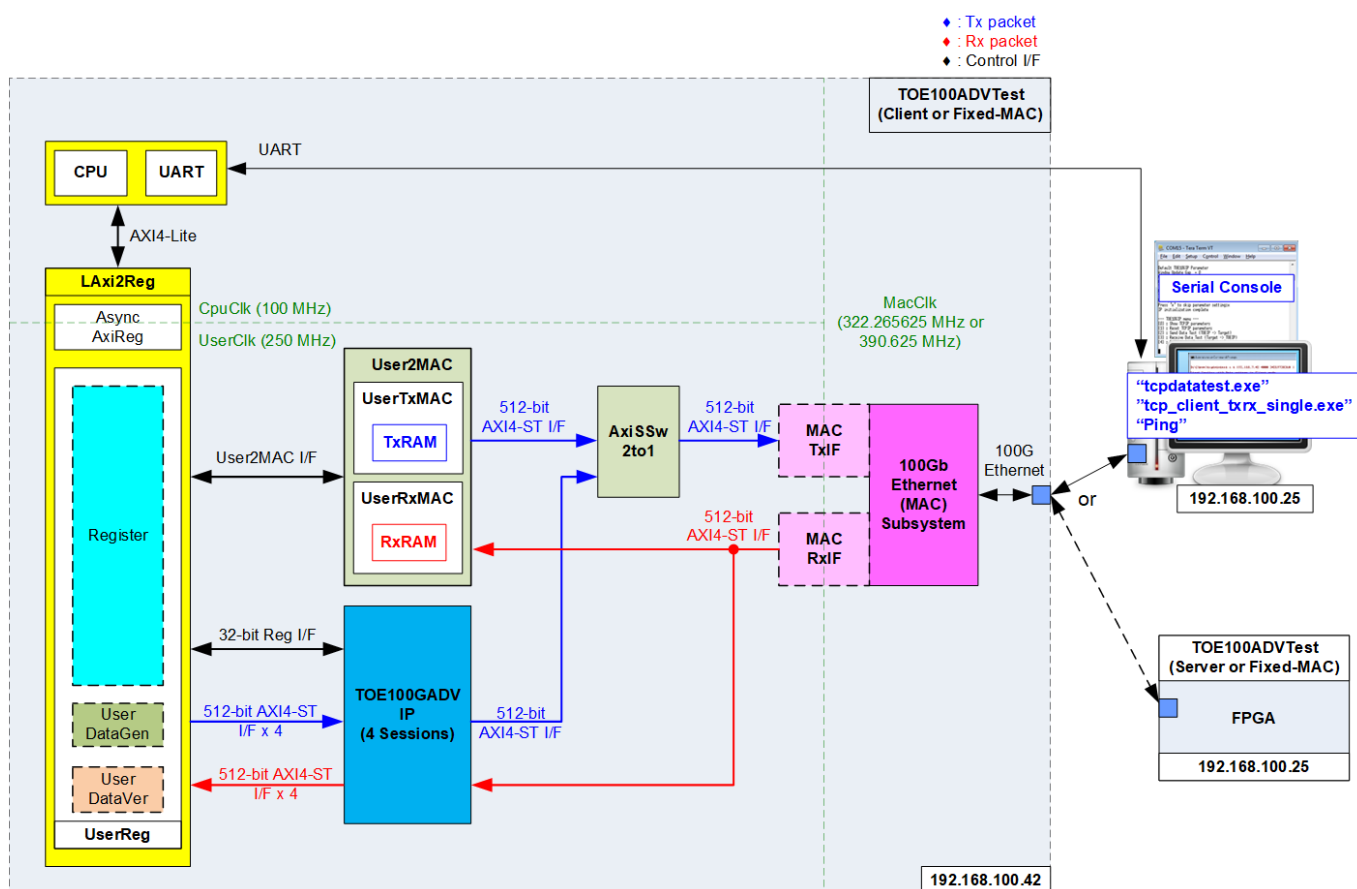


Figure 2-1 Demo block diagram

In the test environment, two devices are utilized for the transfer of 100G 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 and tcp_client_trx_single) 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 TOE100GADV-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 MAC. UserReg serves as the interface for CPU access to TxRAM and RxRAM.

For the high-speed connection, TOE100GADV-IP is integrated to process TCP payload data for four sessions. The CPU configures the parameters of TOE100GADV-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 AxiSSw2to1 module functions as the switch logic, selecting the source of transmitted packets to the Ethernet (MAC) subsystem, which can be User2MAC or TOE100GADV-IP. The receive interface of the Ethernet (MAC) subsystem is directly connected to both User2MAC and TOE100GADV-IP, each incorporating their packet filtering logics to selectively bypass specified packets.

For the Ethernet MAC module, this reference design utilizes the 100G Ethernet MAC IP, a hard IP integrated into Xilinx FPGA. The Ethernet MAC IP core differs between UltraScale+ and Versal devices. On UltraScale+ devices, it is referred to '100G Ethernet Subsystem', featuring a user interface that utilizes a 512-bit AXI4 stream. This interface matches the EMAC I/F of TOE100GADV-IP; however, the operational clock domain of TOE100GADV-IP and the 100G Ethernet Subsystem may typically differ.

On Versal devices, the Ethernet MAC IP is named the '100G Ethernet MAC subsystem', employing a user interface with a 384-bit AXI4 stream. This interface does not match the 512-bit width of TOE100GADV-IP. Consequently, adapter logics, including MACTxIF and MACRxIF, are integrated between TOE100GADV-IP and the specific 100G Ethernet (MAC) Subsystem to facilitate the conversion of interface.

The reference design incorporates three distinct clock domains: CpuClk for the CPU system, MacClk for interfacing with the 100G Ethernet (MAC) Subsystem, and UserClk for the user logic of the TOE100GADV-IP. To facilitate asynchronous signal transfer between CpuClk and UserClk, AsyncAxiReg is specifically designed. Further details about each module within the TOE100GADVTest are provided below.

Note:

- 1) *UserClk can be reconfigured to utilize the same clock as CpuClk, offering a reduction in clock resource usage.*
- 2) *It is recommended to set the UserClk frequency of TOE100G-IP at 220 MHz or higher.*
- 3) *The MacClk frequency for the 100G Ethernet Subsystem (UltraScale+ devices) is 322.266 MHz, while the MacClk frequency of the 100G Ethernet MAC Subsystem (Versal devices) is 390.625 MHz.*

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

The 100G Ethernet (MAC) Subsystem comprises the MAC layer and lower-layer protocol for interfacing with external devices using 100G BASE-SR. This subsystem can be generated through the IP wizard in the Vivado tool, and we delve into two hardware solutions tailored for two different FPGA models.

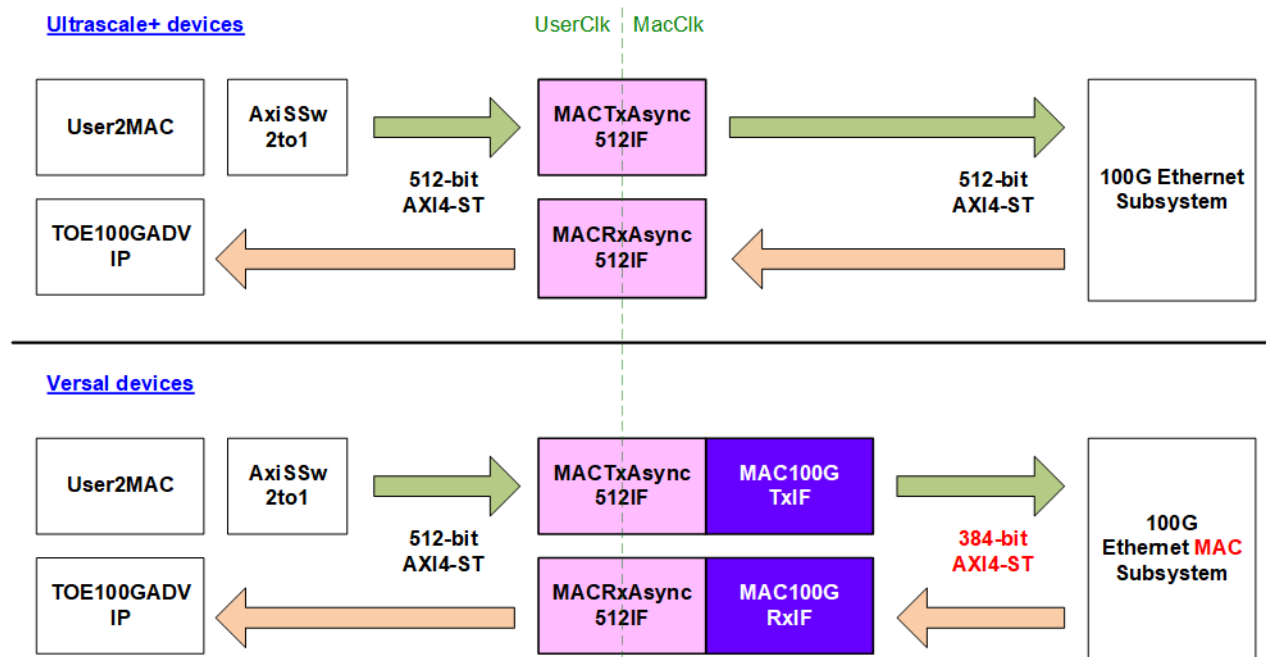


Figure 2-2 Adapter logics of EMAC interface for Xilinx Devices

a) 100G Ethernet Subsystem on UltraScale+ devices

This IP integrates MAC, PCS, and PMA features, along with the Transceiver module. Its user interface is a 512-bit AXI4-stream at 322.266 MHz, necessitating adapter logics (MACTxAsync512IF and MACRxAsync512IF) for seamless connection to TOE100GADV-IP. These logics manage packet transfer across clock domains between UserClk and MacClk. For more detailed information, visit the Xilinx website and check out the “PG203: UltraScale+ Devices Integrated 100G Ethernet Subsystem Product Guide” https://www.xilinx.com/products/intellectual-property/cmec_usplus.html

b) 100G Ethernet MAC Subsystem on Versal devices

This IP includes MAC and PCS features but excludes the Transceiver module. A PMA module must be generated using the IP wizard in the Vivado tool to connect with the 100G Ethernet MAC Subsystem. The user interface of this EMAC can be configured to various modes. In this reference design, we use “Non-Segmented mode with independent clock”, featuring a 384-bit user interface. As this data width is incompatible with the 512-bit width of MACTxAsync512IF and MACRxAsync512IF, two additional adapter logics, MAC100GTxIF and MAC100GRxIF, have been devised to convert the asymmetric width of the interfaces. The 100G EMAC in this mode requires a minimum clock frequency of 390.625 MHz. For more information, visit the Xilinx website and check out the “PG314: Versal Devices Integrated 100G Multirate Ethernet MAC Subsystem Product Guide”. <https://www.xilinx.com/products/intellectual-property/mrmac.html>

2.1.1 MACTxAsync512IF

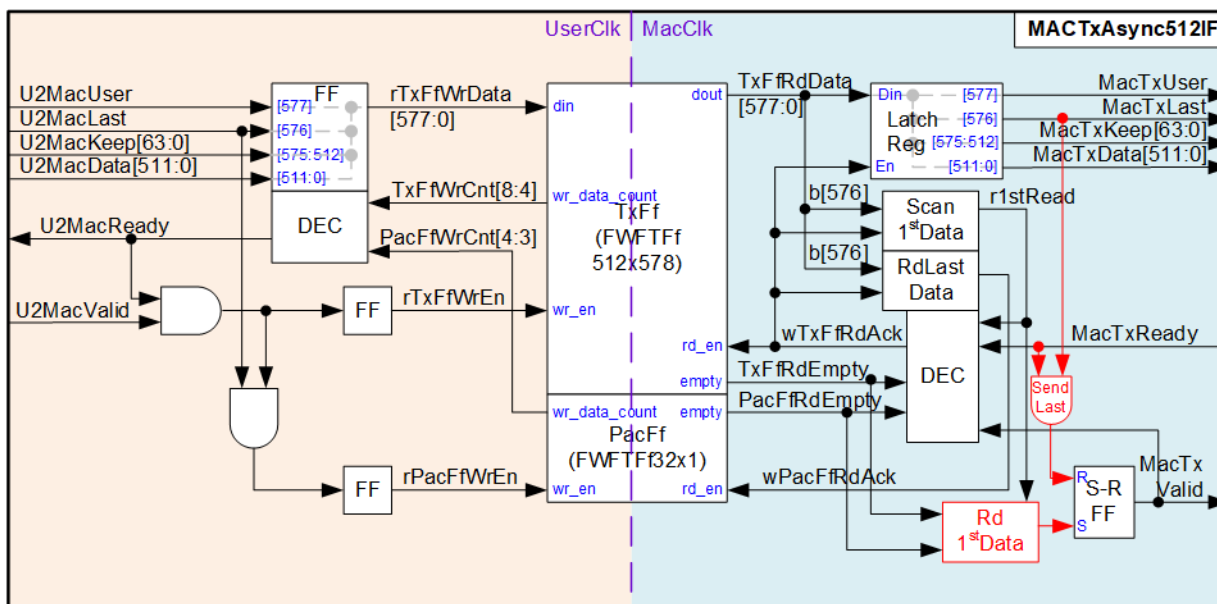


Figure 2-3 MACTxAsync512IF logic diagram

This module serves as an asynchronous adapter facilitating data transfer via a 512-bit AXI4 stream interface from the TOE100GADV-IP (UserClk domain) to the 100G Ethernet (MAC) Subsystem (MacClk domain). Asynchronous crossing is handled through two First-Word Fall-Through (FWFT) FIFOs within the module – specifically, one FIFO buffers the input stream (TxFf), and the other (PacFf) signals the completion of storing all data of each packet in TxFf. Consequently, the adapter comprises two main logic groups – Write and Read operation to and from the FIFOs.

On the Write side, the UserClk frequency typically exceeds the MacClk frequency. After transferring data for a specific duration, the FIFOs reach their capacity, leading to a pause in the write operation. The write data count of both TxFf and PacFf are checked to ensure sufficient space for the subsequent packet transfer. If not, U2MacReady is de-asserted to 0b, halting the incoming stream. To signify the completion of storing each packet in TxFf, a dummy data is written to PacFf when the last data of the packet is stored in TxFf, monitored by U2MacLast being set to 1b.

On the Read side, the operation commences by setting wTxFfRdAck to 1b, initiating the reading of the first data of each packet from TxFf when at least all data of one packet is available in TxFf, monitored by PacFfRdEmpty and TxFfRdEmpty. Additionally, there is no remaining data from the previous packet being transferred. This condition is confirmed by MacTxValid not being set to 1b with MacTxReady is set to 0b. Once the first data has been successfully transferred, the subsequent data within the packet is transmitted until the completion of transferring the last data of the packet.

The status of the next read data, 'r1stRead', is set to 1b if the next read data is the first data of each packet and 0b otherwise. This signal determines the condition for asserting wTxFfRdAck to 1b. Upon the complete transfer of all data for each packet, with the last data read from TxFf, 'wPacFfRdAck' is set to 1b for a single clock cycle to clear the dummy data in PacFf.

'MacTxValid' is generated by S-R FF to maintain a value of 1b during each packet transfer. Consequently, it is asserted when transferring the first data and de-asserted after transferring the last data. For more illustration, the timing diagram of Read operation is depicted in Figure 2-4.

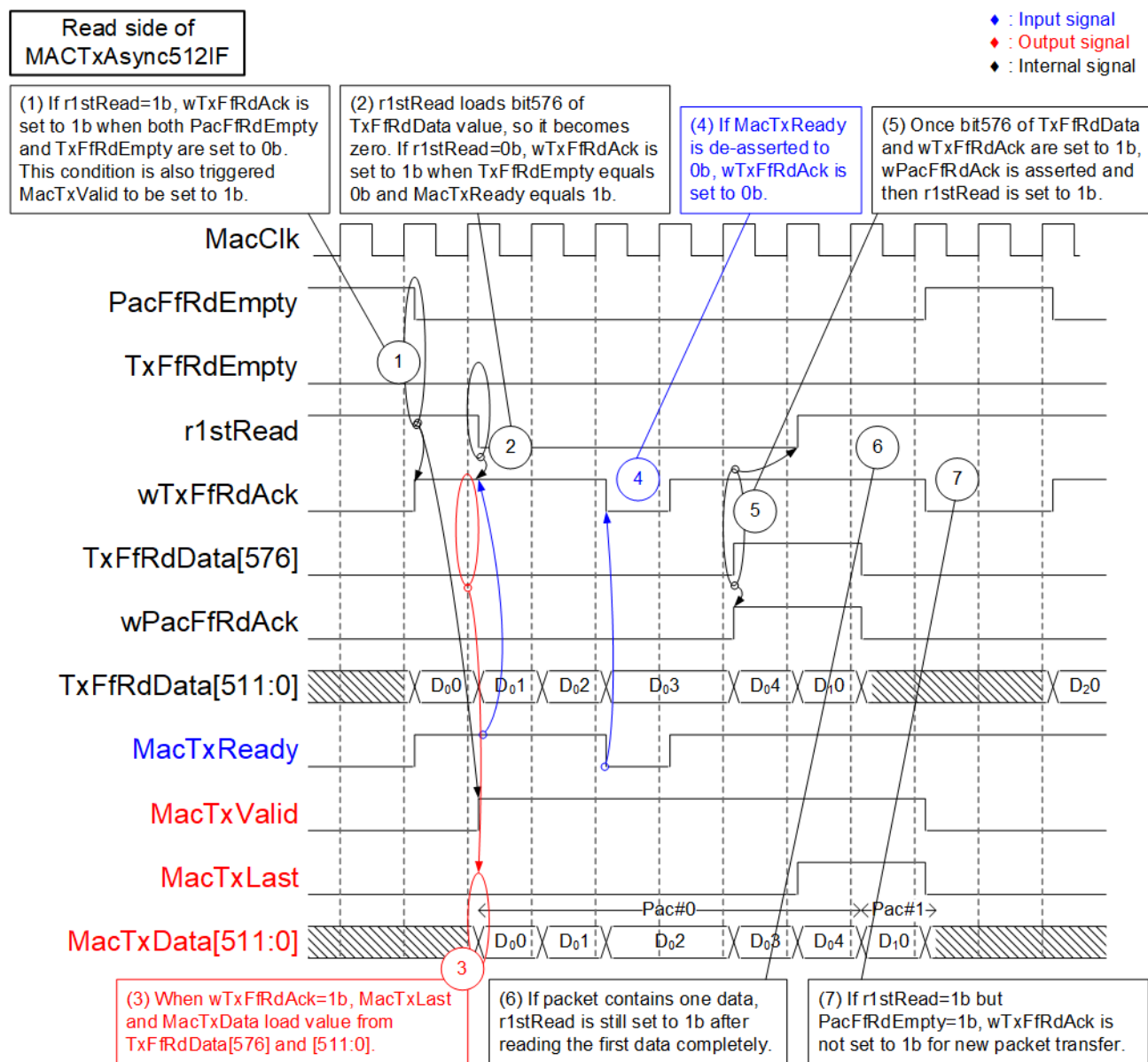


Figure 2-4 Timing diagram of Read operation in MACTxAsync512IF

- 1) A new packet transfer is initiated by three conditions: $r1stRead=1b$, $PacFfRdEmpty=0b$, and $TxFfRdEmpty=0b$, ensuring that the subsequent packet's data is ready for retrieval from $TxFf$. 'wTxFfRdAck' is set to 1b for the first data read, followed by the setting of $MacTxValid$ to 1b in the subsequent clock cycle, thereby initiating the packet transfer. 'MacTxValid' remains at 1b to continuously transfer all data within the packet until the last data is transmitted.
- 2) Assuming the first packet comprises five data ($D_00 - D_04$), $TxFfRdData[576]$, representing the last flag, is set to 0b during the first data read cycle. Subsequently, $r1stRead$ is loaded with the value from $TxFfRdData[576]$, causing it to be set to 0b in the subsequent clock cycle. For reading data other than the first data, indicated by $r1stRead=0b$, $wTxFfRdAck$ is controlled by two conditions: $TxFfRdEmpty=0b$ and $MacTxReady=1b$.
- 3) $TxFf$ is the FWFT type, making the read data ($TxFfRdData$) valid in the same clock cycle in which $wTxFfRdAck$ is set to 1b. Upon the assertion of $wTxFfRdAck$, 512-bit $MacTxData$ and $MacTxLast$ load their value from bits[511:0] and bit[576] of $TxFfRdData$, respectively.
- 4) During packet transfer, if $MacTxReady$ is de-asserted to 0b, $wTxFfRdAck$ is immediately set to 0b, pausing the data read operation and preserving the value of $MacTxData$ and $MacTxLast$.
- 5) Upon reading the last data from $TxFf$, indicated by both $wTxFfRdAck$ and $TxFfRdData[576]$ being set to 1b, $wPacFfRdAck$ is set to 1b for a single clock cycle, flushing one dummy data from $PacFf$. This reduces the total packet count stored in $TxFf$ and re-asserts $r1stRead$ to 1b, preparing for the first data of the subsequent packet.
- 6) After that, the same sequence, starting from step 1, is iterated to forward the new packet. Assuming this packet contains only one data, the last flag is asserted during the first data read cycle. In this condition, $r1stRead$ is not de-asserted after completing the first data read.
- 7) If $TxFf$ stores a portion of the packet, not all the data in the packet, as indicated by $PacFfRdEmpty=1b$ but $TxFfRdEmpty=0b$, the read operation is temporarily paused until the last data of packet is received.

2.1.2 MACRxAsync512IF

MacRxAsync512IF serves as an AXI4-Stream data adapter to cross asynchronous interface from 100G Ethernet (MAC) Subsystem (MacClk) to TOE100GADV-IP (UserClk). It performs asynchronous handling by using one First-Word Fall-Through FIFO. The module consists of a few logic components both in write and read side of the FIFO, including Write Controller, Error Detection, and Read Controller as shown in Figure 2-5.

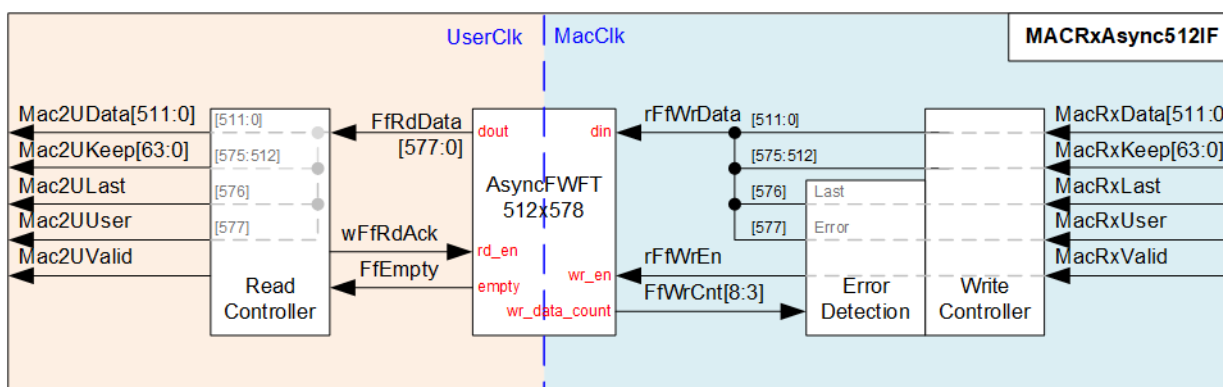


Figure 2-5 MACRxAsync512IF logic diagram

The Write Controller is responsible for synchronizing the input stream with the result of Error Detection module, pack the stream interface to be FIFO write data, and directly write the FIFO in the next step. While Error Detection is incorporated to cancel the write operation when the buffer free space drops below the threshold (less than 8 data). When an error is detected, Error Detection forcedly asserts rFfWrEn, rFfWrData[576], and rFfWrData[577] to 1b, thereby marking the completion of the current packet transmission with an error status. After that, the module waits until the end of this error packet (by detecting assertion of MacRxLast) without writing additional packet data to the FIFO.

The AsyncFWFT512x578 module functions as an asynchronous FWFT FIFO, storing the input stream in the MacClk domain and facilitating the transition to the UserClk domain. It can store 578-bit data which is the whole stream interface, comprising of 512-bit data, a 64-bit keep signal, a 1-bit error flag, and a 1-bit last flag.

The Read Controller reads data from the AsyncFWFT32x578 and forwards it as AXI4-Stream data to the user. The Data is read out and merely forwarded immediately whenever the FIFO contains read data without any flow control because the AXI4-ST interface in the user side excludes the 'Ready' signal.

For more comprehension about Write side of the module, the timing diagram of Write operation is depicted in Figure 2-6.

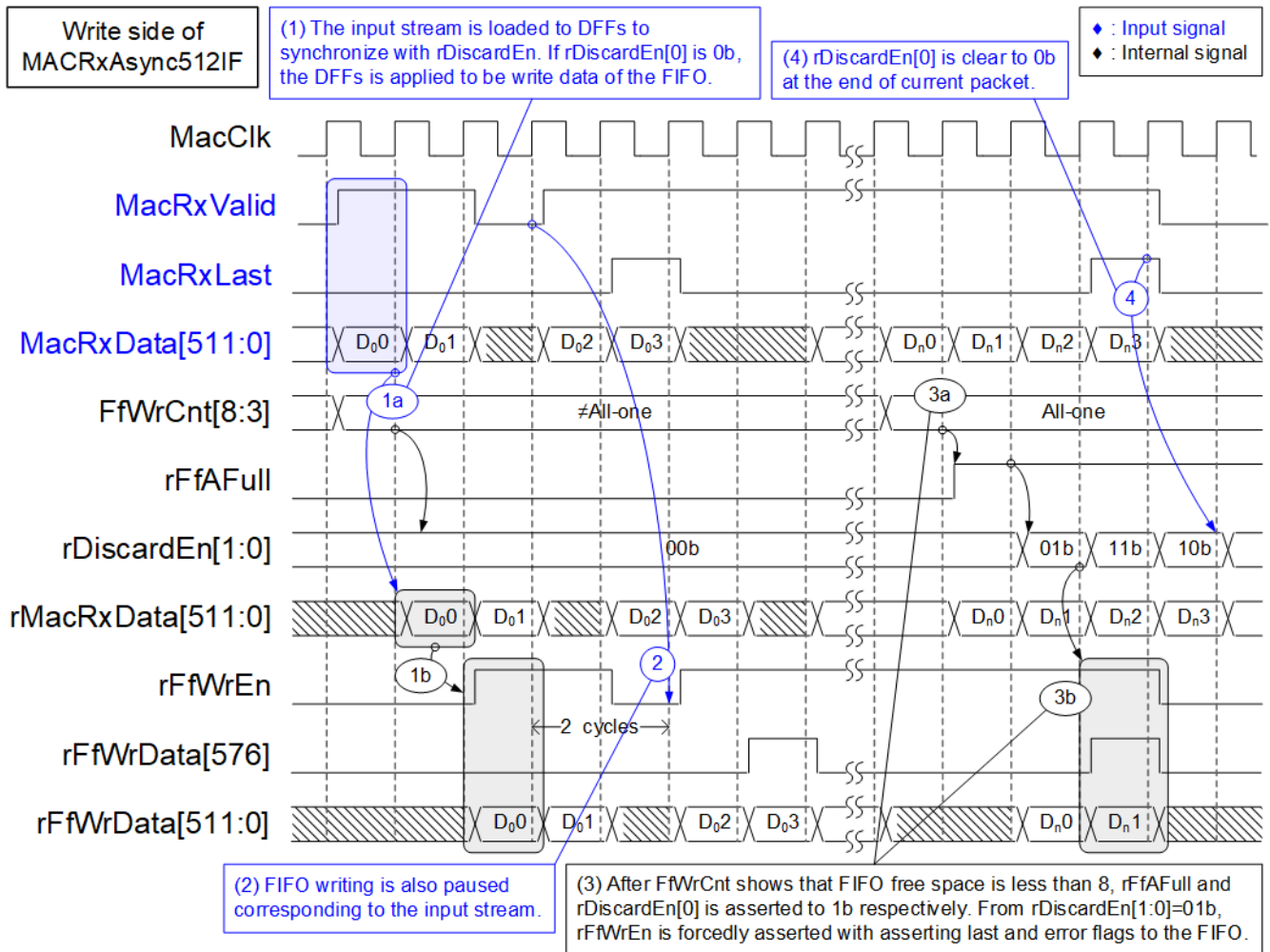


Figure 2-6 Timing diagram of Write operation in MACRxAsync512IF

- 1) During Write side operation, the available space in the FIFO is always monitored using FfWrCnt. The FIFO full condition, indicated by rFfAFull, triggers the assertion of rDiscardEn[0], leading to the dropping of packet. In normal scenario, upon receiving the input stream, the entire stream is loaded to DFFs for synchronization with rDiscardEn[0], such as MacRxData being loaded into rMacRxData. When rDiscardEn[0] is set to 0b, each data stream from the DFFs is written to the FIFO by setting rFfWrEn to 1b.
- 2) Under normal conditions, the value of rFfWrEn is analogous to MacRxValid, accounting for a latency time of two clock cycles for transitioning to 1b or 0b.
- 3) If the FIFO space falls below the threshold value (less than 8), resulting in both rFfFull and rDiscardEn[0] being asserted to 1b, the rising edge of rDiscardEn triggers the termination of current stream being written to the FIFO. This is achieved by setting rFfWrEn to 1b as the last clock cycle, marking the last cycle of this stream. Simultaneously, both the last flag and error flag (Last - rFfWrData[576] and Error - rFfWrData[577]) are set to 1b. After that, the incoming data of the current packet (D_n2 and D_n3) are not written to the FIFO.
- 4) Upon receiving the end of the current packet (MacRxLast is asserted to 1b), the dropped packet function is disabled by de-asserting rDiscardEn[0] to 0b.

2.1.3 MAC100GTxIF

This module serves as an AXI4-Stream converter, transitioning data from 512-bit to 384-bit format for data transfer from the user (TOE100GADV-IP) to the 100G Ethernet MAC Subsystem. In order to facilitate this transfer, a 384-bit register is used to store 128-bit user data for each cycle, which cannot be transmitted to the EMAC. The control signal, rTempCnt, indicates the quantity of unsent data in 128-bit units, stored in 384-bit internal register (rTempData). Four distinct values are assigned to signify the data amount: 000b (No data), 001b (one 128-bit data), 011b (two 128-bit data), and 111b (three 128-bit data or full). The output data sent to EMAC is a mixed signal which combines user data (U2MACData) with the 384-bit rTempData, controlled by rTempCnt. Timing diagram to show more details of MAC100GTxIF is shown in Figure 2-7.

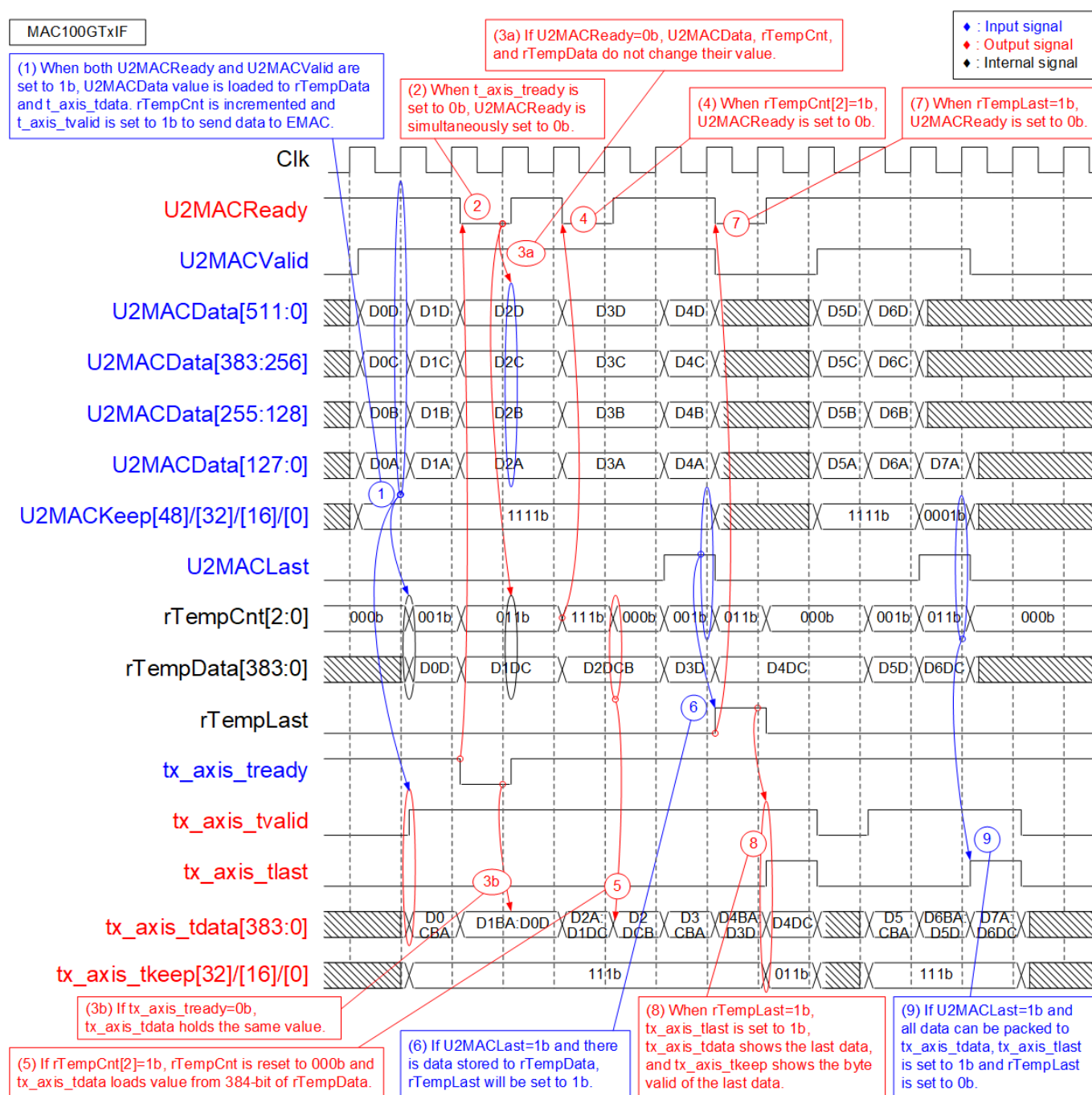


Figure 2-7 MAC100GTx timing diagram

- 1) Upon receiving the first data from the user (U2MACValid=1b and U2MACReady=1b while rTempCnt=000b), the module transfers 384-bit user data (U2MACData) to EMAC. 'tx_axis_tvalid' is asserted to 1b, and tx_axis_tdata loads the 384-bit data from U2MACData. If this data is not the last, the upper 128-bit unsent data is stored in the internal register (rTempData), and rTempCnt increments according to the sequence: 000b -> 001b -> 011b -> 111b. Additionally, tx_axis_tkeep is asserted to all ones to transfer the 384-bit data to EMAC.
- 2) When EMAC is not ready, tx_axis_tready is de-asserted to 0b. U2MACReady is also de-asserted to 0b, pausing the transmission of user data.
- 3) If tx_axis_tready and U2MACReady are de-asserted to 0b, the output signals to EMAC (tx_axis_tvalid, tx_axis_tlast, tx_axis_tdata, and tx_axis_tkeep) and the input signals from the user (U2MACValid, U2MACData, U2MACKeep, and U2MACLast) must retain the same values until the ready signals are re-asserted to 1b to accept the current data.
- 4) When the 384-bit register (rTempData) stores three 128-bit data, and rTempCnt equals 111b (indicating a full condition), U2MACReady is de-asserted to 0b to pause user data transmission. Subsequently, the 384-bit data from rTempData is flushed to EMAC.
- 5) 'tx_axis_tdata' loads 384-bit data from rTempData, and rTempCnt is reset to 000b, signifying no remaining unsent data stored in rTempData.
- 6) The last user data is transmitted by asserting U2MACLast to 1b. U2MACKeep is read to determine the number of valid bytes in the last data. Additionally, rTempCnt is read to check the amount of unsent data. In the provided example, one 128-bit data is stored in rTempCnt, and 512-bit user data is received, requiring the storage of two 128-bit data in rTempCnt. In such cases, rTempLast is asserted to 1b to store the unsent last data.
Note: Step 9) provides an example when the last user data is received, but all data can be transferred to EMAC without storing any data in rTempData.
- 7) rTempLast is asserted to 1b when the last user data is stored in rTempData. Simultaneously, U2MACReady is de-asserted to 0b, pausing user data transmission.
- 8) This step illustrates a scenario when two 128-bit data are stored in rTempData, and 128-bit last data is transmitted by user. Consequently, the total data, which comprises three 128-bit data, can be transferred to tx_axis_tdata by asserting tx_axis_tlast to 1b. In this case, no data is remained in rTempData, and rTempLast is not asserted to 1b.

2.1.4 MAC100GRxIF

This module functions as an AXI4-Stream converter, converting data from 384-bit to 512-bit for transmission from the 100G Ethernet MAC Subsystem to the user (TOE100GADV-IP). The design incorporates a Latch register to store unsend data that has not been transmitted to the user. Three counters facilitate data realignment: 'wRx128bDataCnt' indicates the received data amount from EMAC (1, 2, or 3), 'rLatDataCnt' shows the unsend data amount received from EMAC (0-3), and 'wRxTotalDataCnt' indicates the sum of received and unsend data (wRx128bDataCnt + rLatDataCnt), ranging from 1 to 6.

When wRxTotalDataCnt is 4 or greater (5 or 6), 512-bit data is packed and transmitted to the user. However, the last transmitted data may be less than 512 bits, controlled by the byte enable value (MAC2UKeep).

The second counter (rLatDataCnt) is updated under various conditions.

- 1) Upon receiving the first data with no unsend data stored in rDataLat (rLatDataCnt=0), all bits of the first data are loaded into rDataLat. Therefore, rLatDataCnt must equal the received data amount from EMAC (wRx128bDataCnt or wRxTotalDataCnt, which is the same value when rLatDataCnt=0).
- 2) When the total data amount (wRxTotalDataCnt) equals or exceeds 4, indicating the transmission of a 512-bit data to TOE100GADV-IP, the unsend data amount (rLatDataCnt) is reduced by 4 (wRxTotalDataCnt – 4).
- 3) Upon the transmission of the last data with no new packet received, the Latch register is now in an empty state. Consequently, rLatDataCnt is reset to 0.
- 4) A special case arises when the last data is transmitted while the first data of a new packet is received. This scenario is a combination of condition 1) and 3). Therefore, the unsend data amount is equal to the received data amount in the new packet (wRx128bDataCnt).

The 384-bit latch register (rDataLat) utilizes rLatDataCnt to determine the maximum amount of received data from EMAC that must be retained in the next cycle.

- 1) When rLatDataCnt = 0, the requirement is to retain three 128-bit new data (rx_axis_tdata[384:0]).
- 2) When rLatDataCnt = 3, it packs one 128-bit new data with three 128-bit previous data (rDataLat[383:0]). Therefore, two 128-bit new data (rx_axis_tdata[384:128]) must be retained in rDataLat.
- 3) When rLatDataCnt = 2, two 128-bit new data are packed with two 128-bit previous data (rDataLat[255:0]). Therefore, one 128-bit data (rx_axis_tdata[384:256]) must be retained in rDataLat.
- 4) When rLatDataCnt = 1, all new data can be packed with one 128-bit previous data (rDataLat[127:0]). In this case, no data is retained in rDataLat.

To facilitate the transfer of the last data from EMAC to the user, two behaviors are considered.

- 1) If all the last data from EMAC can be packed with rDataLat (wRxTotalDataCnt ≤ 4), the last data will be transmitted to the user in the next cycle.
- 2) In cases where wRxTotalDataCnt for the last data exceeds 4, two cycles are required to transmit all data – 512-bit data during the 1st cycle and the remaining data during the 2nd cycle. To support this feature, rExLast is designed to latch the last flag of EMAC for transmitting the last data during the 2nd cycle.

Timing diagram to show MAC100GRxIF operation is shown in Figure 2-8.

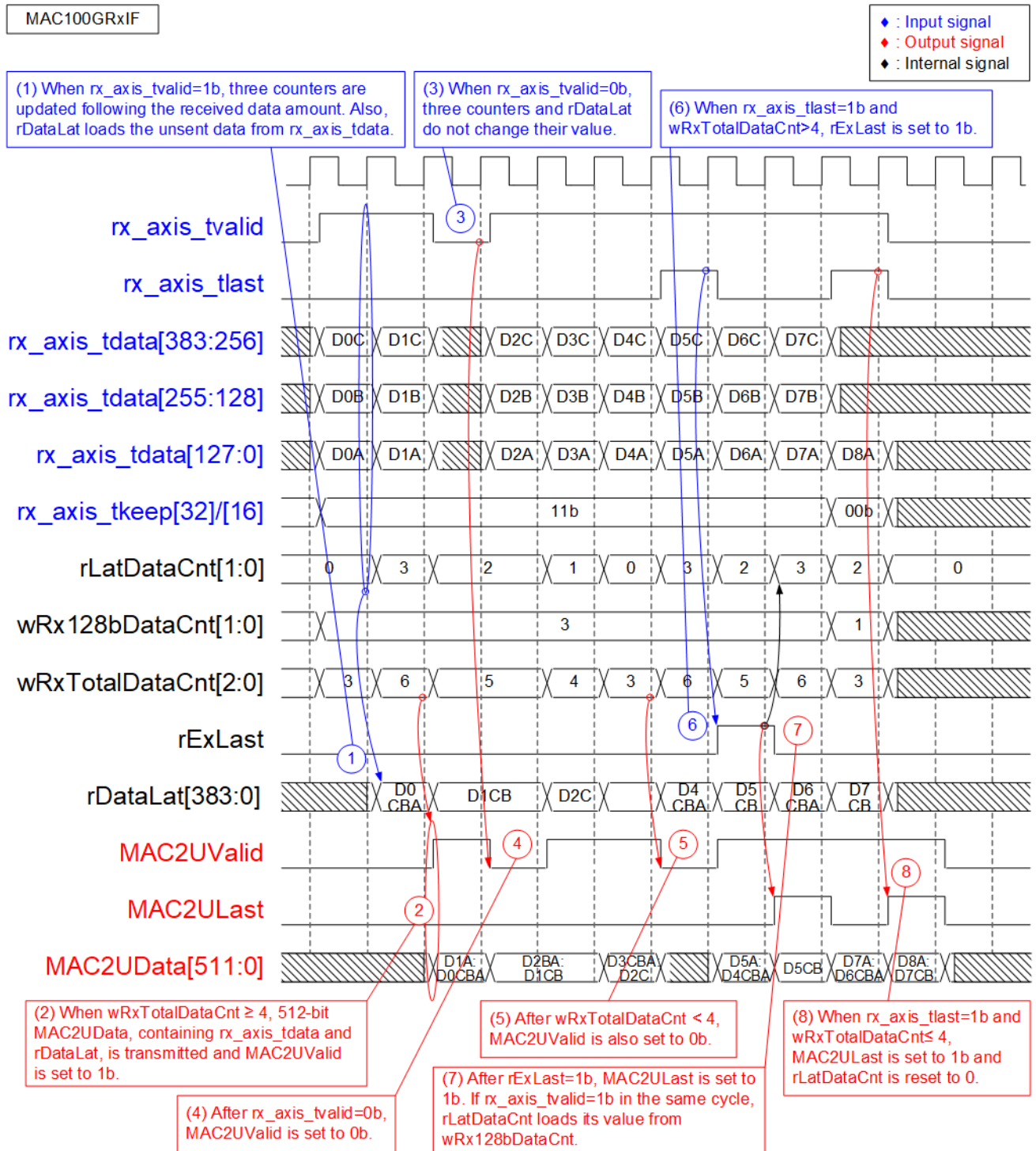


Figure 2-8 MAC100GRxIF Timing diagram

- 1) Upon receiving new 384-bit data from EMAC, `wRx128bDataCnt` equals 3. If the previous clock cycle is Idle (`rLatDataCnt=0`), `wRxTotalDataCnt` is equal to 3 (0+3). When `rLatDataCnt` is 0, the entire 384-bit data is loaded into `rDataLat`. Since `wRxTotalDataCnt` is less than 4, no data is transmitted to the MAC2U I/F.
- 2) Subsequently, when another 384-bit data arrives from EMAC with `rLatDataCnt` equal to 3 (the amount of data stored in `rDataLat` in the previous clock cycle), `wRxTotalDataCnt` becomes 6 (3 + 3), sufficient to transmit data to MAC2U I/F. `MAC2UValid` is asserted to 1b, facilitating the transmission of 512-bit `M2UData`. `M2UData` loads three 128-bit data (`D0A`, `D0B`, and `D0C`) from `rDataLat` and one 128-bit data from `rx_axis_tdata` (`D1A`). Consequently, two 128-bit data (`D1B` and `D1C`) remain unsent and are stored in `rDataLat`. `rLatDataCnt` is updated to 2.
- 3) If EMAC de-asserts `rx_axis_tvalid` to 0b because it is not ready to transmit new data, the three counters (`rLatDataCnt`, `wRx128bDataCnt`, and `wRxTotalDataCnt`), along with `rDataLat`, maintain the same values, awaiting more data from EMAC.
- 4) In the event that EMAC pauses data transmission by de-asserting `rx_axis_tvalid` to 0b, `MAC2UValid` is de-asserted to 0b in the subsequent clock.
- 5) During the first cycle of every four cycles to receive 384-bit data from EMAC, when `rLatDataCnt` is equal to 0 and `wRxTotalDataCnt` is less than 4, `MAC2UValid` is de-asserted to 0b, pausing data transmission to the user.
- 6) Upon receiving the last data from EMAC (`rx_axis_tlast=1b` and `rx_axis_tvalid=1b`) and when `wRxTotalDataCnt` in that cycle is more than 4 (5 or 6), the latch flag to store the last signal (`rExLast`) is asserted to 1b. Simultaneously, 512-bit data is transferred to the user, while the remaining last data is transferred in the subsequent cycle.
- 7) After `rExLast` is asserted to 1b, `MAC2ULast` is also asserted to 1b to transmit the remaining last data stored in `rDataLat`. If the first data of the new packet is promptly transferred from EMAC, it is loaded into `rDataLat`, and `rLatDataCnt` is set to the amount of 128-bit data in the first cycle.
- 8) This step illustrates an example of sending the last data without asserting `rExLast`. When `rx_axis_tlast` is asserted to 1b and `wRxTotalDataCnt` is less than or equal to 4, the last data can be packed and transferred to the user in the subsequent cycle. Therefore, `rExLast` is not asserted to 1b, and `rLatDataCnt` is reset to 0.

2.2 AxiSSw2to1

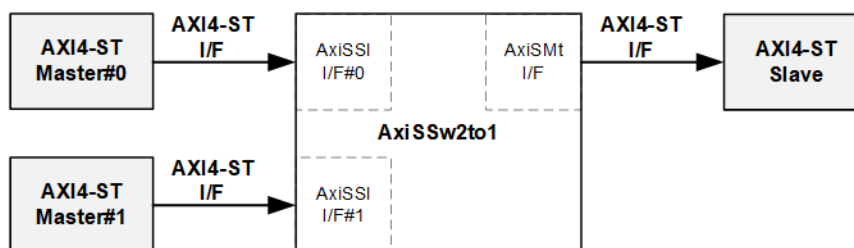


Figure 2-9 AxiSSw2to1 interface

This module serves as a 2-to-1 switch logic for AXI4-ST interface, facilitating the transfer of transmitted data from User2MAC or TOE100GADV-IP to the Ethernet (MAC) subsystem. AxiSSw2to1 incorporates configurable parameters allowing the selection of data and user signal widths. In this reference design, the data width for User2MAC and TOE100GADV-IP is set at 512 bits, while the data width for the user signal is 1 bit.

Conceptually, AxiSSw2to1 operates by transferring data from two Masters (Ch#0 and Ch#1) to one Slave. In cases where both channels request data transfer simultaneously, AxiSSw2to1 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 AXISSw2to1 logic to select the active AxiSSi 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 2-10.

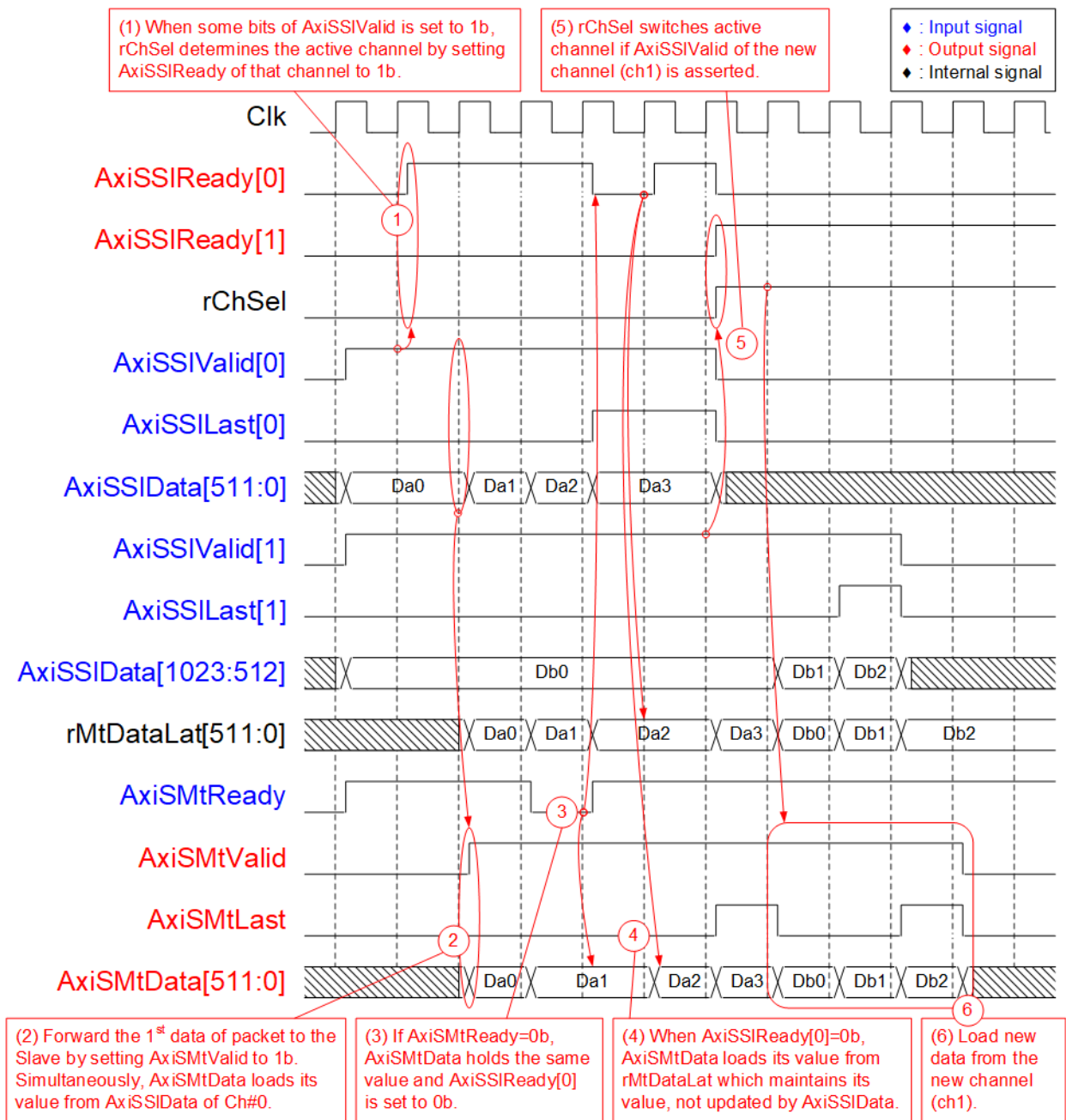


Figure 2-10 AxiSSw2to1 timing diagram

- 1) When two users simultaneously initiate the transmission of a new packet by asserting AxiSSIVValid 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 2-10, Ch#0 is selected, prompting the assertion of AxiSSIRReady 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 AxiSSILast[0] (indicating the end-of-packet) and AxiSSIData[511:0] (512-bit data), are loaded as output signals to the external Slave via the Master I/F (AxiSMtLast and AxiSMtData, respectively). Additionally, AxiSMtValid 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 AxiSMtReady to 0b, all output signals of the Master I/F maintain the same values. Also, AxiSSIRReady for the active channel is de-asserted to 0b, preserving the input signals from the Master.
- 4) Upon the Slave re-asserting AxiSMtReady to accept data, the output signals to the Slave load the next values from the internal latch register (rMtDataLat). The internal latch register stores data from the active source when AxiSSIRReady 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 AxiSSIVValid for another channel is asserted, rChSel switches its value. In Figure 2-10, the next active channel becomes Ch#1 (rChSel=1b), facilitating the acceptance of data from Ch#1.
- 6) The input signals (AxiSSILast and AxiSSIData) from the active channel (Ch#1) are forwarded to become the output signals of the Slave (AxiSMtLast and AxiSMtData) until the final data of packet is completely transferred.

2.3 TOE100GADV-IP

The TOE100GADV-IP implements TCP/IP offloading engine for handling four TCP sessions with the same target. The user data interface utilizes a 512-bit AXI4 stream interface. The control interface is used to configure the network parameters, send the command request, and monitors the operation status. The Ethernet MAC interface utilizes a 512-bit AXI4 stream interface. Further information of the IP can be found on our website.

2.4 User2MAC

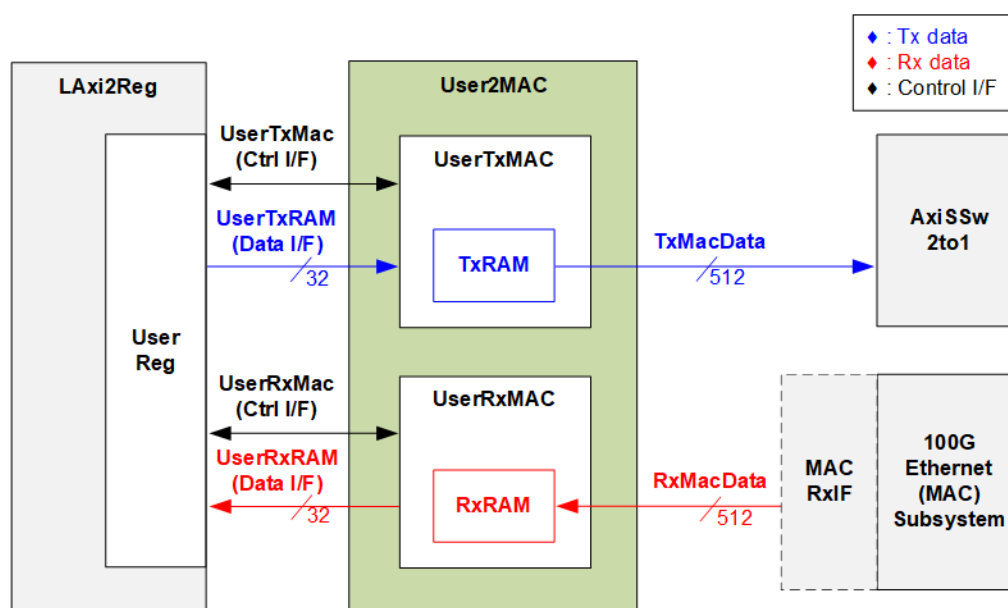


Figure 2-11 User2MAC block diagram

User2MAC is designed for transferring Ethernet packets for low-speed connection. The reference design incorporates the Ping command, using the ICMP protocol, to measure round-trip time, with an ICMP echo reply packet generated upon receiving an ICMP echo request packet. The CPU utilizes LAXi2Reg to create and decode the Ethernet packet, with a data bus width of 32 bits on the LAXi2Reg side and 512 bits on the MAC I/F side.

For operating in both transmission and reception, User2MAC comprises two modules: UserTxMAC and UserRxMAC. UserTxMAC includes TxRAM, where the CPU prepares and stores transmitted Ethernet packets. Meanwhile, UserRxMAC features RxRAM to store Ethernet packets received from EMAC. Prior to storing a packet in RxRAM, a filtering logic checks Ethernet header, ensuring only valid packets are stored, while invalid ones are rejected. Subsequently, the CPU reads from RxRAM to decode the stored packet. Further details about UserTxMAC and UserRxMAC are provided below.

2.4.1 UserTxMAC

UserTxMAC includes 64 x 512-bit simple dual port RAM to store transmitted packets, written by CPU via the UserTxRam write I/F. The CPU sets the packet size (UserTxLen), and upon asserting the request (UserTxReq), the logic initiates forwarding the packet, read from TxRAM, to EMAC. The transmit interface of EMAC is a 512-bit AXI4 stream, which may de-assert 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 2-12.

Note: The UserTxRam Write I/F with the CPU utilizes a 32-bit data width, while the TxRAM data width is 512 bits. Therefore, a decoder is implemented to create a write byte enable, allowing the CPU to write only specific bytes of the 512-bit data bus of TxRAM.

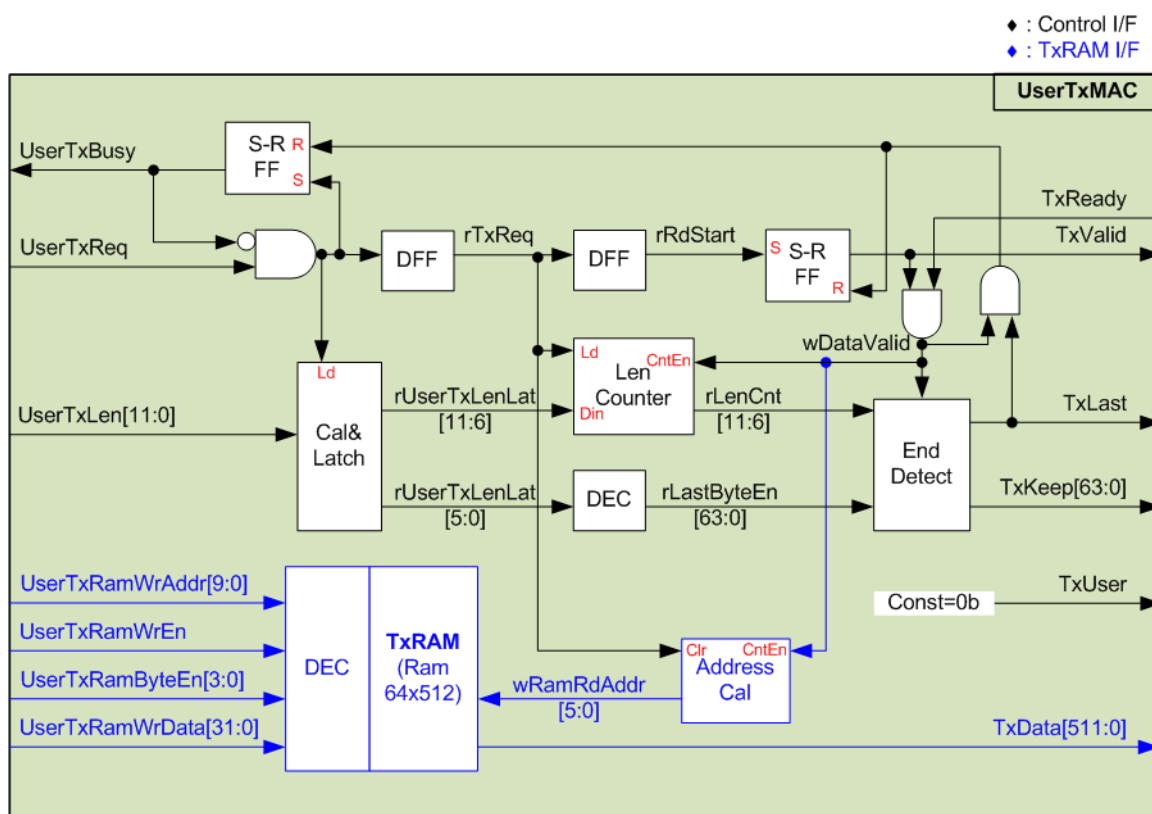


Figure 2-12 UserTxMAC Logic Diagram

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 a transmitted packet and writes it to TxRAM. The first data is written at address#0 (UserTxRamWrAddr=0). The maximum size of the transmitted packet size is 4 KB, matching the size of TxRAM.
Note: TxRAM incorporates a byte enable to enable CPU to write data using a byte unit.
- 3) The CPU sets UserTxLen to specify the transmit packet size in byte units. Additionally, UserTxReq is asserted to 1b to initiate data transmission.
- 4) Subsequently, the request signal is loaded into several logics and a DFF chain. UserTxBusy is asserted to 1b to indicate to the user that the operation is in progress. The total transfer size (UserTxLen) is loaded into internal logic, split into two parts. The first part is the amount of 512-bit data calculated using UserTxLen[11:6]. The value is rounded up if the size is not aligned to 512-bit. The second part is bit[5:0], which is latched to create the byte enable of the final data of a packet (rLastByteEn and TxKeep) using a decoder.
- 5) When the start flag is asserted (rRdStart), the first data is read from TxRAM, and the data valid (TxValid) is asserted to 1b. The read address (wRamRdAddr) is up-counted to transfer the next data from TxRAM after completing each data transfer (TxValid=1b and TxReady=1b). Additionally, the Length counter (rLenCnt) is down-counted upon finishing the transfer of each data to check the last data position.
- 6) When rLenCnt=1 or the next data is the final data of a packet, the last flag (TxLast) is asserted to 1b. Also, the byte enable (TxKeep) loads the last value from rLastByteEn. TxKeep is set to all ones to send 512-bit data when the data is not the final data of the packet.
- 7) After the final data is transferred, the busy flag is de-asserted to 0b to complete the current transfer.

2.4.2 UserRxMAC

UserRxMAC performs three distinct operations to validate received packets and store valid packets in RxRAM, which is a 64 x 512-bit simple dual port RAM. As a result, the logic within UserRxMAC can be categorized into three groups.

Firstly, there is logic dedicated to verifying the 38-byte header (byte#0 – byte#37) of each received packet. The user can set the expected value and mask bit to enable data comparison. Only packets with the correct header are accepted.

Secondly, there is logic to check the enable flag from the user and the free space in RxMacFf. The packet will be rejected if the user disables this module or if the FIFO does not have sufficient space. RxMacFf is used to store the end address of RxRAM after completing storing the received packet. Consequently, the CPU determines the received packet size based on the end address.

Thirdly, there is logic dedicated to storing the received packet in RxRAM. Additional details about the internal logic design of UserRxMAC are illustrated in Figure 2-13.

Note: The UserRxRam Read I/F with the CPU employs a 32-bit data width, while RxRAM has a data width of 512 bits. Therefore, a 16-to-1 Mux is integrated to select 32-bit data from the 512-bit data.

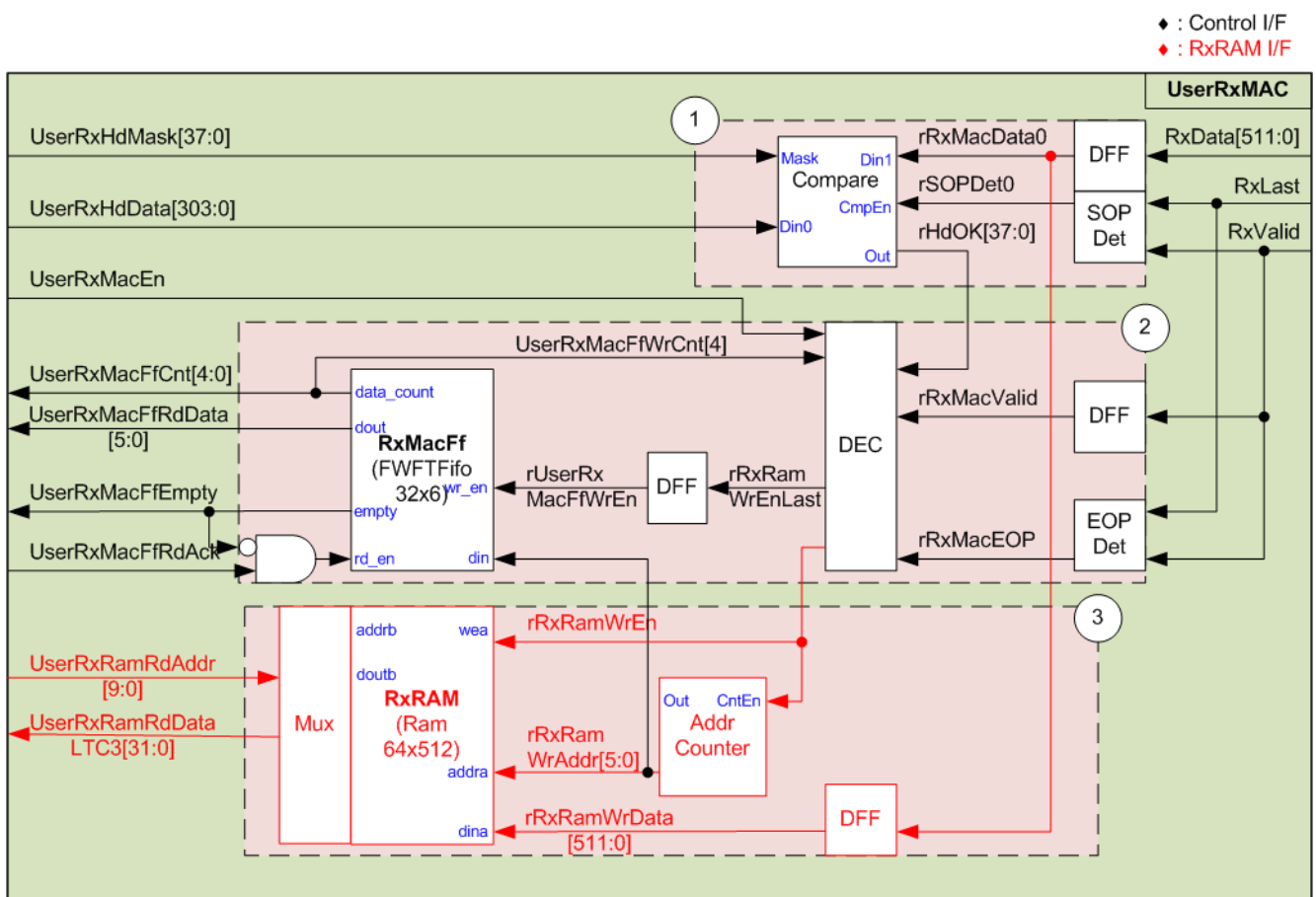


Figure 2-13 UserRxMAC Logic Diagram

As shown in Figure 2-13, Block (1) comprises the logic responsible for verifying the 38-byte header of each packet. Block (2) is dedicated to storing the end address of RxRAM, and Block (3) manages the storage of the received packet. The details of UserRxMAC operation upon receiving a packet are elaborated as follows.

- 1) 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) Subsequently, two signals are additional read – the enable flag from user (UserRxMacEn) and RxMacFf data counter (UserRxMacFfWrCnt). It needs to confirm that 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 finishing storing each packet in RxRAM. Therefore, EOPDet is designed to assert 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) Lastly, the valid packet is stored to RxRAM by asserting the write enable to RxRAM (rRxRamWrEn) when the data is received (RxValid=1b). The address counter is incremented after each 512-bit data is stored to 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 read the last address using the UserRxMacFfRdData signal, which is valid for reading due to RxMacFf being a FWFT 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 512-bit data. Therefore, the CPU firmware must convert the 512-bit address stored in RxMacFf to a 32-bit address before starting data reading from RxRAM.

2.5 CPU and Peripherals

The CPU system uses a 32-bit AXI4-Lite bus as the interface to access peripherals such as the Timer and 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 AXI4-Lite bus standard. LAXi2Reg module, as shown in Figure 2-14, is designed to connect the CPU system via the AXI4-Lite interface, in compliance with the standard.

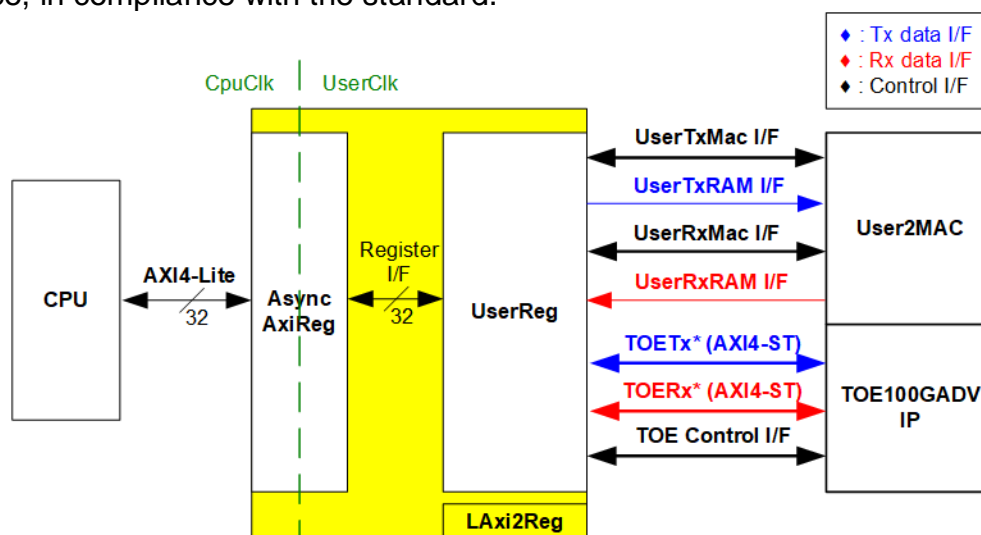


Figure 2-14 LAXi2Reg block diagram

LAXi2Reg consists of AsyncAxiReg and UserReg. AsyncAxiReg converts AXI4-Lite signals into a simple register interface with a 32-bit data bus size, similar to the AXI4-Lite 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 TOE100GADV-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 TOE100GADV-IP employs an AXI4-ST interface, facilitated through TOETx I/F and TOERx I/F. Additional details regarding AsyncAxiReg and UserReg are provided below.

2.5.1 AsyncAxiReg

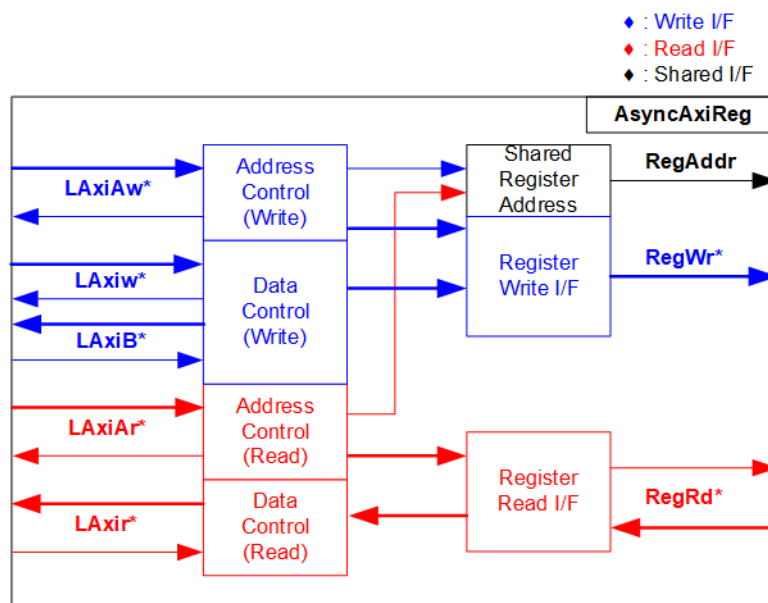


Figure 2-15 AsyncAxiReg interface

The signal on AXI4-Lite bus interface can be grouped into five groups, i.e., LAXiAw* (Write address channel), LAXiw* (Write data channel), LAXiB* (Write response channel), LAXiAr* (Read address channel), and LAXir* (Read data channel). More details to build custom logic for AXI4-Lite bus is described in following document.

https://github.com/Architech-Silica/Designing-a-Custom-AXI-Slave-Peripheral/blob/master/designing_a_custom_axi_slave_rev1.pdf

According to AXI4-Lite standard, the write channel and read channel operate independently for both control and data interfaces. Therefore, the logic within AsyncAxiReg to interface with AXI4-Lite bus is divided into four groups, i.e., Write control logic, Write data logic, Read control logic, and Read data logic, as shown in the left side of Figure 2-15. The Write control I/F and Write data I/F of the AXI4-Lite bus are latched and transferred to become the Write register interface with clock domain crossing registers. Similarly, the Read control I/F of AXI4-Lite bus is latched and transferred to the Read register interface, while Read data is returned from Register interface to AXI4-Lite via clock domain crossing registers. In the Register interface, RegAddr is a shared signal for write and read access, so it loads the value from LAXiAw for write access or LAXiAr for read access.

The Register interface is compatible with single-port RAM interface for write transaction. The read transaction of the Register interface has been slightly modified from RAM interface by adding the RdReq and RdValid signals to control read latency time. The address of Register interface is shared for both write and read transactions, so user cannot write and read the register at the same time. The timing diagram of the Register interface is shown in Figure 2-16.

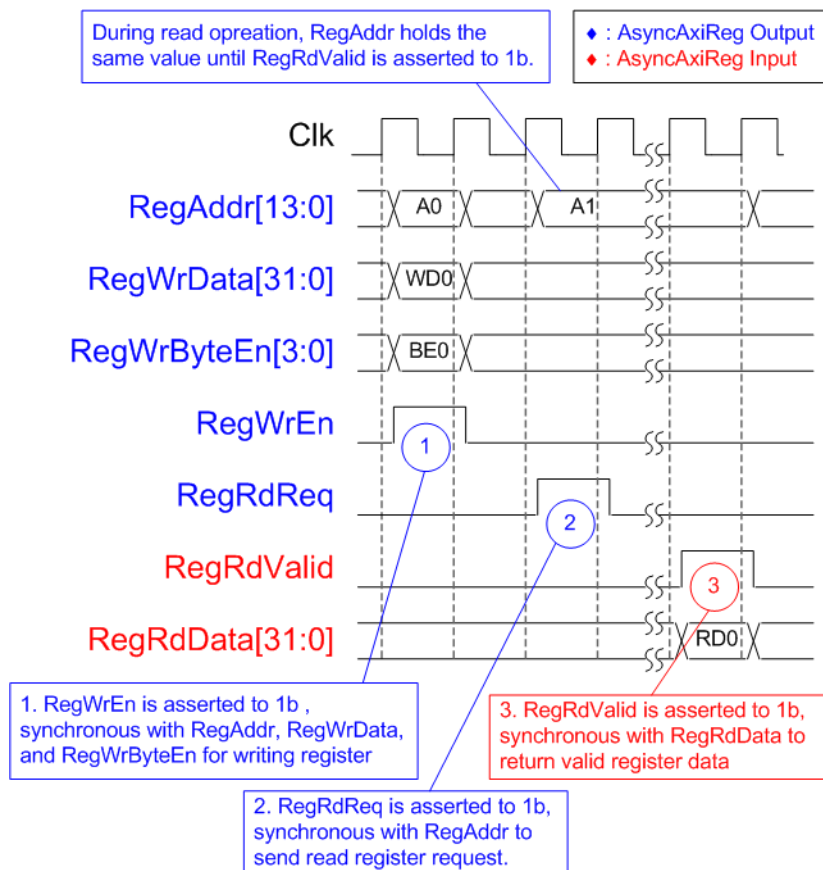


Figure 2-16 Register interface timing diagram

- 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, AsyncAxiReg 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, AsyncAxiReg forwards the read value to the LAxir* interface.

2.5.2 UserReg

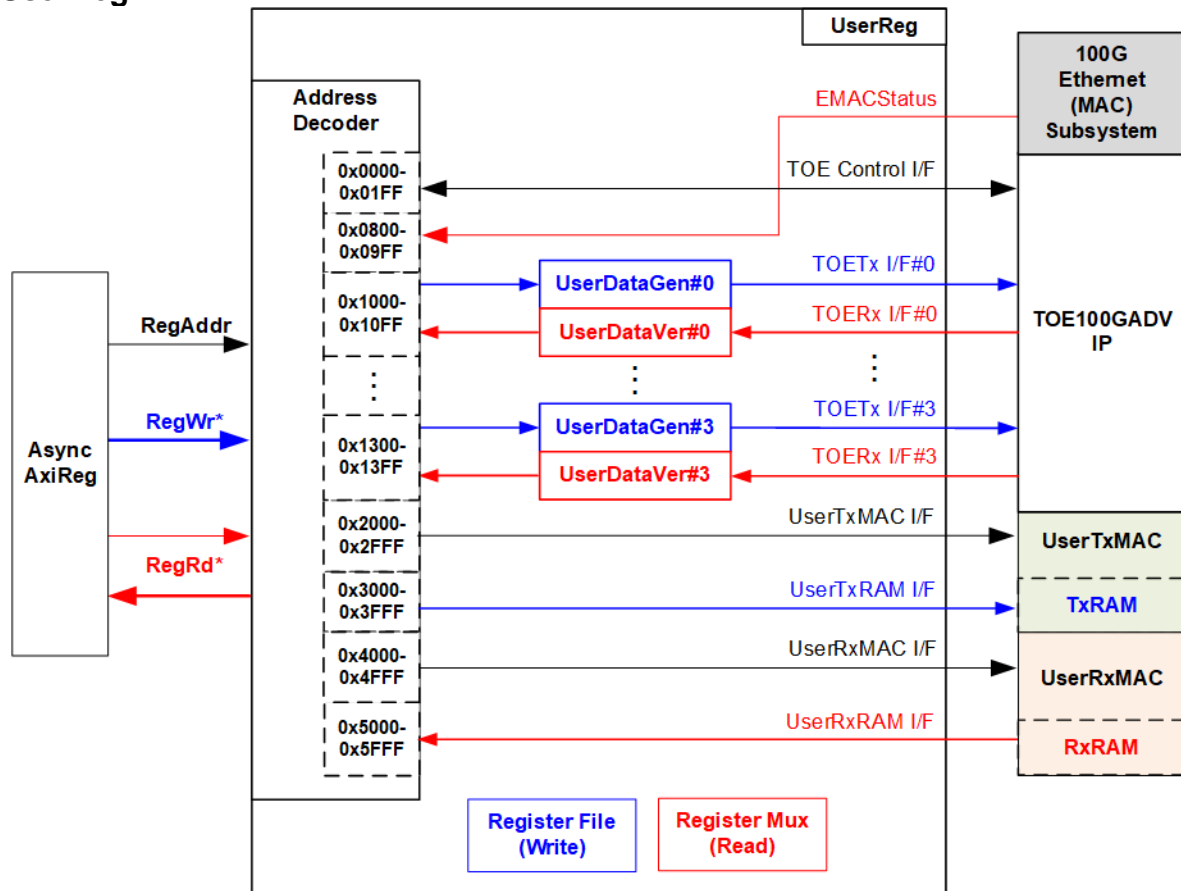


Figure 2-17 UserReg block diagram

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 2-17, the address range mapped to UserReg is divided into seven areas.

- 1) 0x0000 – 0x01FF: Control and status signals of TOE100GADV-IP
- 2) 0x0800 – 0x09FF: Status signal of EMAC
- 3) 0x1000 – 0x013FF: UserDataGen and UserDataVer signals for transferring data with 4 sessions of TOE100GADV-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 in Table 2-1.

To read register, multiple multiplexers are utilized to select data from each module, leading to increased real 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 Wr/Rd	Register Name (Label in the "toe100gadvttest.c")	Description
(BA+0x0000) – (BA+0x01FF): Control and Status of TOE100GADV-IP		
Further information of each TOE100GADV-IP I/O signals is described in the datasheet.		
(BA+0x0000) – (BA+0x00FF): Hardware system signals		
BA+0x0000 Wr/Rd	Hardware reset (HW_RST_INTREG)	[0]: Set to 1b to reset TOE100GADV-IP, all UserDataGen modules, and all UserDataVer modules.
BA+0x0084 Rd	Ethernet MAC status (EMAC_STS_INTREG)	[0]: Ethernet MAC link status (0b-Link down, 1b-Link up) [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)
(BA+0x0100) – (BA+0x013F): Common parameters and status signals		
BA+0x0100 Rd	TOE IP version (TOE_VER_INTREG)	[31:0]: Mapped to IPVersion[31:0] of TOE100GADV-IP
BA+0x0104 Rd	TOE initial finish flag (TOE_INF_INTREG)	[0]: Mapped to InitFinish of TOE100GADV-IP
BA+0x0110 Wr/Rd	Source MAC Address Low (TOE_SML_INTREG)	[31:0]: Mapped to SrcMacAddr[31:0] of TOE100GADV-IP
BA+0x0114 Wr/Rd	Source MAC Address High (TOE_SMH_INTREG)	[15:0]: Mapped to SrcMacAddr[47:32] of TOE100GADV-IP
BA+0x0118 Wr/Rd	Dest MAC Address In Low (TOE_DMIL_INTREG)	[31:0]: Mapped to DstMacAddr[31:0] of TOE100GADV-IP
BA+0x011C Wr/Rd	Dest MAC Address In High (TOE_DMIH_INTREG)	[15:0]: Mapped to DstMacAddr[47:32] of TOE100GADV-IP
BA+0x0120 Wr/Rd	Source IP Address (TOE_SIP_INTREG)	[31:0]: Mapped to SrcIPAddr[31:0] of TOE100GADV-IP
BA+0x0124 Wr/Rd	Dest IP Address (TOE_DIP_INTREG)	[31:0]: Mapped to DstIPAddr[31:0] of TOE100GADV-IP
BA+0x0128 Wr/Rd	Dest MAC Mode (TOE_DMM_INTREG)	[1:0]: Mapped to DstMacMode[1:0] of TOE100GADV-IP
BA+0x012C Wr/Rd	Window Threshold (TOE_WIN_INTREG)	[9:0]: Mapped to WindowThres[9:0] of TOE100GADV-IP
BA+0x0130 Wr/Rd	TCP Control Timeout (TOE_TCT_INTREG)	[31:0]: Mapped to TCPCtlTimeOutSet[31:0] of TOE100GADV-IP
BA+0x0134 Wr/Rd	TCP Receive Timeout (TOE_TRT_INTREG)	[23:0]: Mapped to TCPRxTimeOutSet[23:0] of TOE100GADV-IP
BA+0x0138 Rd	Dest MAC Address Out Low (TOE_DMOL_INTREG)	[31:0]: Mapped to DstMacAddrOut [31:0] of TOE100GADV-IP
BA+0x013C Rd	Dest MAC Address Out High (TOE_DMOH_INTREG)	[15:0]: Mapped to DstMacAddrOut [47:32] of TOE100GADV-IP

Address	Register Name	Description
Wr/Rd	(Label in the "toe100gadvtest.c")	
(BA+0x0140) – (BA+0x017F): Session control and status signals		
BA+0x0140	Source Port Number	[15:0]: Mapped to TCPSrcPort[15:0] of TOE100GADV-IP
Wr/Rd	(TOE_SPN_INTREG)	
BA+0x0144	Dest Port Number	[15:0]: Mapped to TCPDstPort[15:0] of TOE100GADV-IP
Wr/Rd	(TOE_SPN_INTREG)	
BA+0x0148	TCP Last Mode	[1:0]: Mapped to TCPLastMode[1:0] of TOE100GADV-IP
Wr/Rd	(TOE_LMD_INTREG)	
BA+0x014C	TCP Command	Wr
Wr/Rd	(TOE_CMD_INTREG)	[3:0]: Set value to TCPConnCmd[3:0] of TOE100GADV-IP When this register is written, the connection request (TCPConnReq) is asserted to initiate the TOE100GADV-IP operation. Rd [3:0]: The latest value of TCPConnCmd[3:0] set to TOE100GADV-IP [8]: Mapped to TCPConnReady of TOE100GADV-IP
BA+0x0150	TCP Connection ON	[3:0]: Mapped to TCPConnOn[3:0] of TOE100GADV-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 TCPPrInt is triggered by common functions (TCPPrIntStatus[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 TCPPrInt is triggered by session#0 functions (TCPPrIntStatus[31:16] is non-zero). [17]: Set to 1b when TCPPrInt is triggered by session#1 functions (TCPPrIntStatus[47:32] is non-zero). [18]: Set to 1b when TCPPrInt is triggered by session#2 functions (TCPPrIntStatus[63:48] is non-zero). [19]: Set to 1b when TCPPrInt is triggered by session#3 functions (TCPPrIntStatus[79:64] is non-zero). [24]: Set to 1b when TCPPrstInt is triggered by session#0 functions (TCPPrstIntStatus[31:16] is non-zero). [25]: Set to 1b when TCPPrstInt is triggered by session#1 functions (TCPPrstIntStatus[47:32] is non-zero). [26]: Set to 1b when TCPPrstInt is triggered by session#2 functions (TCPPrstIntStatus[63:48] is non-zero). [27]: Set to 1b when TCPPrstInt is triggered by session#3 functions (TCPPrstIntStatus[79:64] is non-zero).
Wr/Rd	(TOE_INT_INTREG)	

Address	Register Name	Description
Wr/Rd	(Label in the "toe100gadvttest.c")	
(BA+0x0180) – (BA+0x01FF): Session#0 – Session#3 status signals		
BA+0x0180 Rd	TCP Conn Status#0 Low (TOE_TCS0L_INTREG)	[31:0]: Mapped to TCPConnStatus[31:0] of TOE100GADV-IP (session#0)
BA+0x0184 Rd	TCP Conn Status#0 High (TOE_TCS0H_INTREG)	[31:0]: Mapped to TCPConnStatus[63:32] of TOE100GADV-IP (session#0)
BA+0x0188 Rd	TOE Transmit Status#0 (TOE_TTS0_INTREG)	[31:0]: Mapped to TOETxStat0[31:0] of TOE100GADV-IP (session#0 status)
BA+0x018C Rd	TOE Receive Status#0 (TOE_TRS0_INTREG)	[31:0]: Mapped to TOERxStat0[31:0] of TOE100GADV-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.
(BA+0x1000) – (BA+0x13FF): UserDataGen and UserDataVer interface		
(BA+0x1000) – (BA+0x107F): UserDataGen#0 control/status		
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 16000 MB/s (250 MHz x 512-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.
(BA+0x1080) – (BA+0x10FF): UserDataVer#0 control/status		
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 16000 MB/s (250 MHz x 512-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 "toe100gadvtest.c")	
BA+0x1100 – BA+0x13FF: UserDataGen and UserDataVer interface for session#1 – session #3		
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.	
BA+0x2000 – BA+0x3FFF: UserTxMAC		
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)	
BA+0x4000 – BA+0x5FFF: UserRxMAC		
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 FWFT 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[5:0] of RXMAC_FF_INTREG. The user sets this bit to 1b once to flush one data from FIFO.
BA+0x4064	RxMacFf of UserRxMAC	[5:0]: 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 TOE100GADV-IP. Each UserDataGen instance is dedicated to transmitting data for one TCP session data to TOE100GADV-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 512-bit AXI4 stream interface.

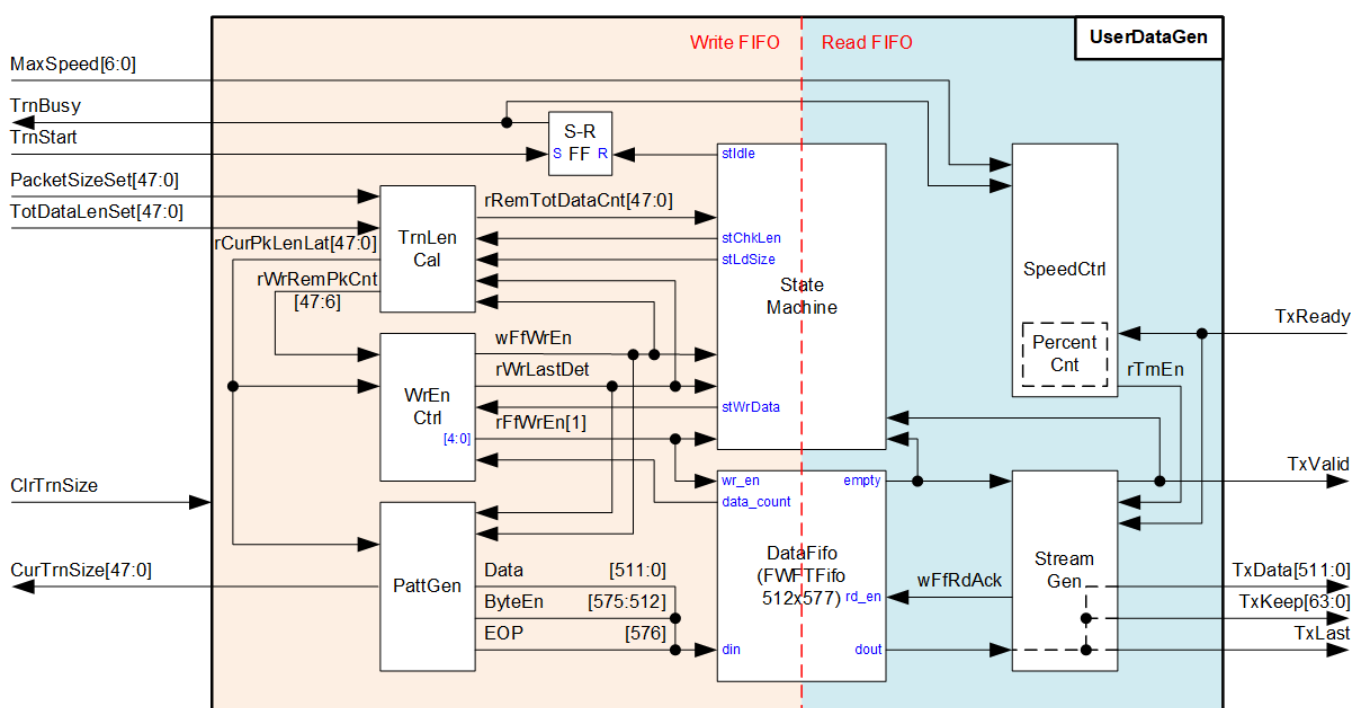


Figure 2-18 UserDataGen block diagram

Figure 2-18 illustrates the block diagram of UserDataGen. The module incorporates a Fast-Word Fall-Through FIFO named DataFifo, serving as a buffer for the generated test data before streaming it out as AXI4-ST. As a result, the logic of UserDataGen is categorized into two groups based on this FIFO – the Write and Read sides of the FIFO.

On the FIFO Write side, a state machine is designed to generate flow control signals for writing each data packet to DataFifo. 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.

The first is TrnLenCal, a counter group providing information to other logics about the necessary transfer size, described as below.

- rRemTotDataCnt: Remaining transfer size in byte units initialized from TotDataLenSet and decreases after a packet is written to the FIFO.
- rCurPkLenLat: Indicates the data size of each packet in byte units. Every packet size is typically equal to PacketSizeSet, except for the last packet. When 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. The initial cycle amount corresponds to the rCurPkLenLat value.

The second is the write enable controller (WrEnCtrl). This logic manages the assertion of FIFO write enable (wFfWrEn and rFfWrEn, sourced by wFfWrEn with specific latency time for signal synchronization) and the last 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.

Lastly, PattGen is designed to generate the 32-bit incremental test data, which is finally packed with byte enable (ByteEn) and last flag (EOP) to become the write data of DataFifo. The initial value of test data is zero and resets to zero after the user asserts a new start flag (TrnStart) or ClrTrnSize. CurTrnSize, for user monitoring, is also handled by the PattGen logic.

Comparing to the Write function, the FIFO Read side comprises there are simply two logic groups: SpeedCtrl and StreamGen. SpeedCtrl is designed to introduce a pause time for transmitting the data stream. This logic, based on counter, asserts rTrnEn signal within a specific period defined by the MaxSpeed value, every 100 transfer cycles. Subsequently, rTrnEn is de-asserted to 0b until the 100-cycle counter is triggered for the new 100-cycle round.

StreamGen is responsible for reading data from DataFifo only when rTrnEn is asserted. The readiness of the read data (FIFO Empty) and the stream interface (TxReady) are also considered in deciding the FIFO reading. Finally, the FIFO read data is formed to become the output AXI4 stream of UserDataGen. Further operation details of UserDataGen are depicted as timing diagram presented in Figure 2-19 and Figure 2-20.

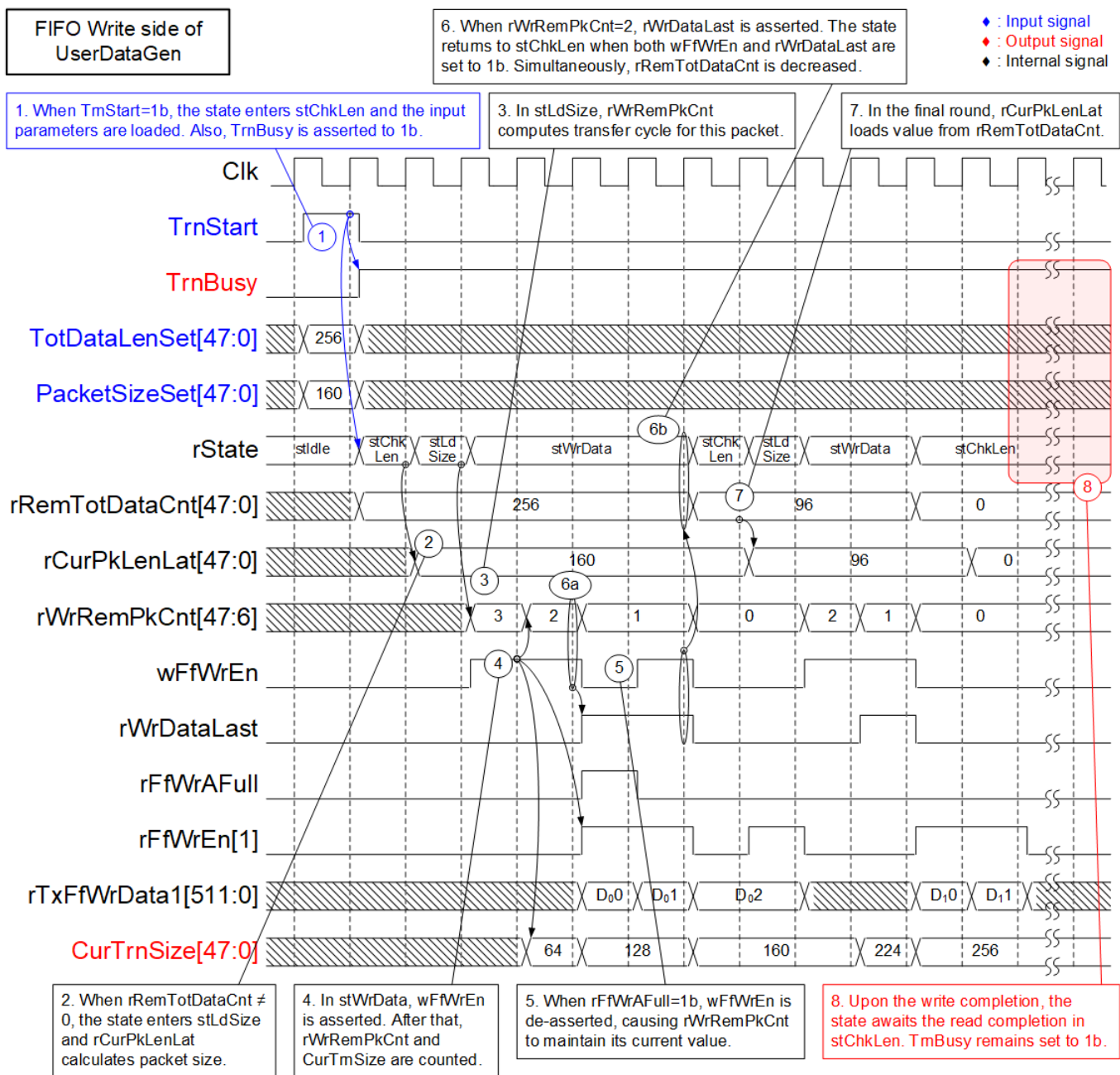


Figure 2-19 Timing diagram of FIFO Write side in UserDataGen

- 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 DataFifo. Subsequently, the state enters stWrData.
- 4) 'stWrData' is the period for writing one packet of data to DataFIFO. The WrEnCtrl logic asserts wFfWrEn to write DataFifo if the FIFO has sufficient free space (rFfAFull=0b). When wFfWrEn is asserted, rWrRemPkCnt is decreased, and FIFO write enable (rFfWrEn[1]) is asserted in the next two clock cycles. Additionally, CurTrnSize is incremented to inform the user about the current transfer size.
- 5) If DataFifo has insufficient free space, rFfWrAFull is set to 1b, resulting in the immediate de-assertion of wFfWrEn. Also, rFfWrEn[1] is de-asserted to 0b in the next two 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 FIFO 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.

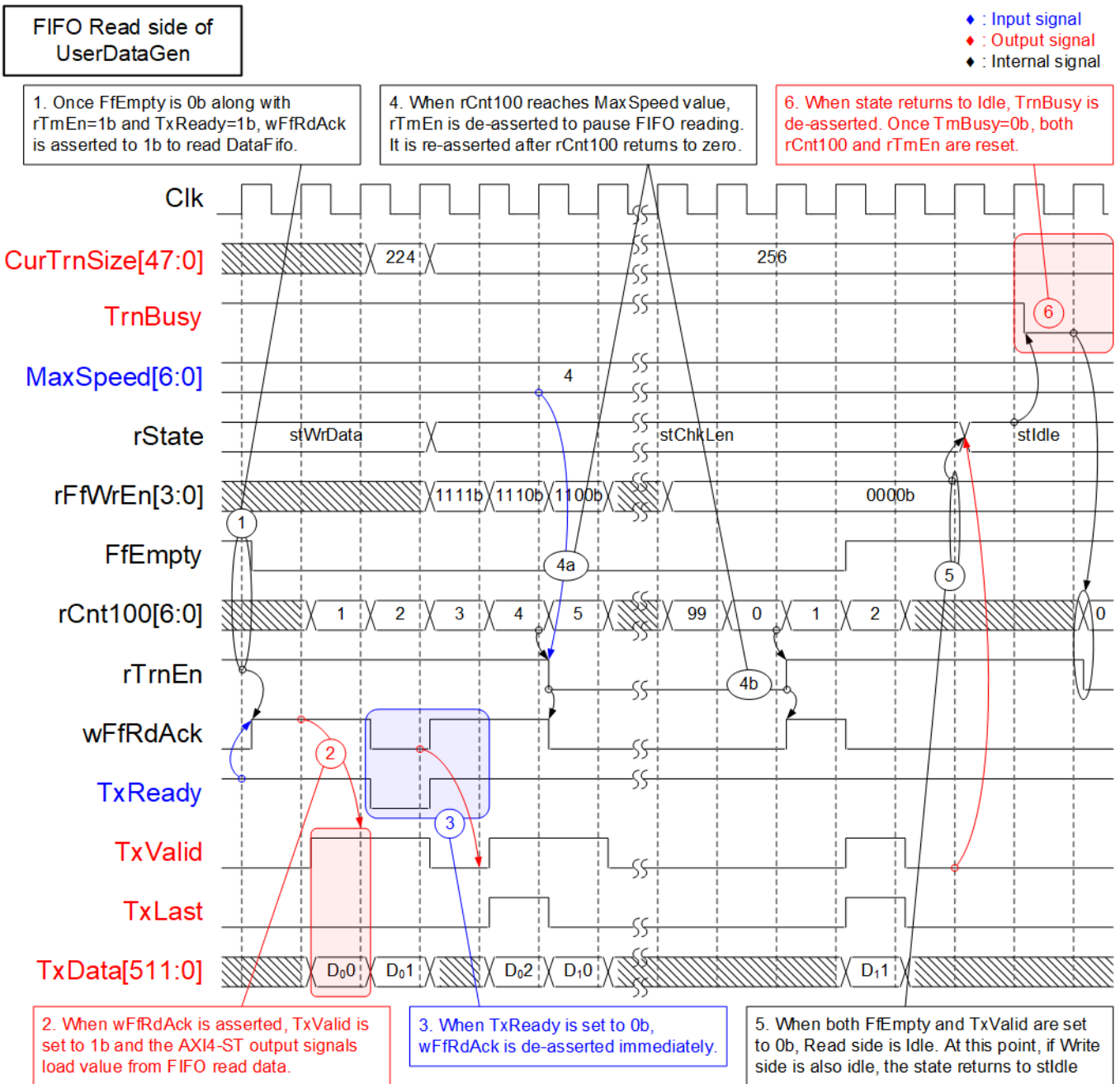


Figure 2-20 Timing diagram of FIFO Read side in UserDataGen

- 1) The read FIFO operation is initiated when three conditions are met: DataFifo must have data (FfEmpty=0b), the transfer speed should not exceed the set value (rTrnEn=1b), and TOE100GADV-IP must be ready to receive data (TxReady=1b). If all three conditions are met, wFfRdAck is asserted to read data out from DataFifo.
- 2) Once wFfRdAck is asserted to 1b, FIFO read data is loaded to become the output signals of the AXI4 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 (FfEmpty=0b) and no packet transfer in AXI4-ST I/F (TxValid=0b), the read operation is completed. If the write operation is also completed, with rFfWrEn[3:0] 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 TOE100GADV-IP. Four UserDataVer modules are specifically allocated for receiving data from four TCP sessions of TOE100GADV-IP. If the user sets the enable flag (RecvEn) to 1b, this module asserts the ready signal to receive the data stream via the AXI4 stream and subsequently verifies it. In the absence of the RecvEn assertion, the ready signal is de-asserted, resulting in the pause of data transmission from TOE100GADV-IP.

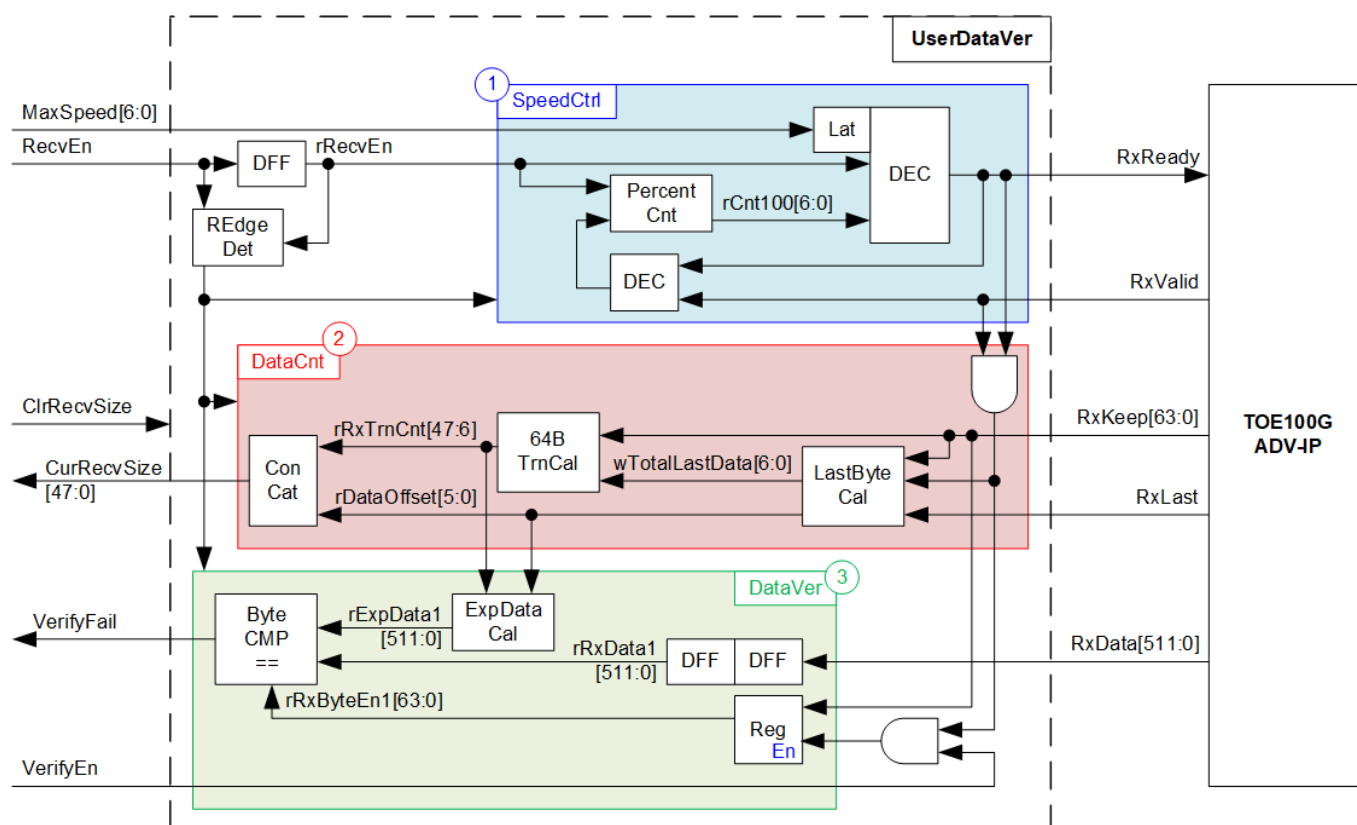


Figure 2-21 UserDataVer block diagram

As shown in Figure 2-21, UserDataVer incorporates three operations: Speed Control, represented by block (1), for controlling transfer speed; Data Counter, denoted as block (2), for counting the received data amount; and Data Verification, designated as block (3), for verifying 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 as follows.

The Speed Control 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 TOE100GADV-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.

The Data Counter incorporates a counter to display the total amount of data received from TOE100ADV-IP, employing two distinct calculation units. The first unit, 64BTrnCal, is dedicated to counting the received data amount in 64-byte units. Generally, 64-byte data is received in every transfer cycle, except for the last cycle which may contain 1-64 bytes of data. Consequently, the second calculation unit, LastByteCal, is designed to compute the amount of remaining received data in byte units by checking RxReady, RxValid, RxLast, and RxKeep. The output of LastByteCal, denoted as wTotalLastDataCnt, is calculated by aggregating the byte offset from the preceding packet transmission and the number of valid bytes in the last transfer cycle of the current packet transmission. If the result exceeds 64 bytes, wTotalLastDataCnt[6] is set to 1b, causing the 64BTrnCal to increment its result, rRxTrnCnt, by one. The remaining bytes that do not align with 64 bytes are stored in the rDataOffset signal. Consequently, the total amount of received data, CurRevSize, can be computed from rRxTrnCnt[47:6] and rDataOffset[5:0]. These values can be cleared by the rising edge of the new RecvEn or by setting the clear flag (ClrRecvSize) to 1b.

The Data Verification incorporates the 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 the user activates the receive enable flag (RecvEn) and verification enable flag (VerifyEn), each byte of the expected data is compared with the received data stream (rRxData1). In the event of data verification error, the Fail flag (VerifyFail) is set to 1b until to notify the user of a mis-matched in the data stream. This Fail flag remains set to 1b until the user asserts the new RecvEn or ClrRecvSize.

3 CPU firmware and Test software

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.

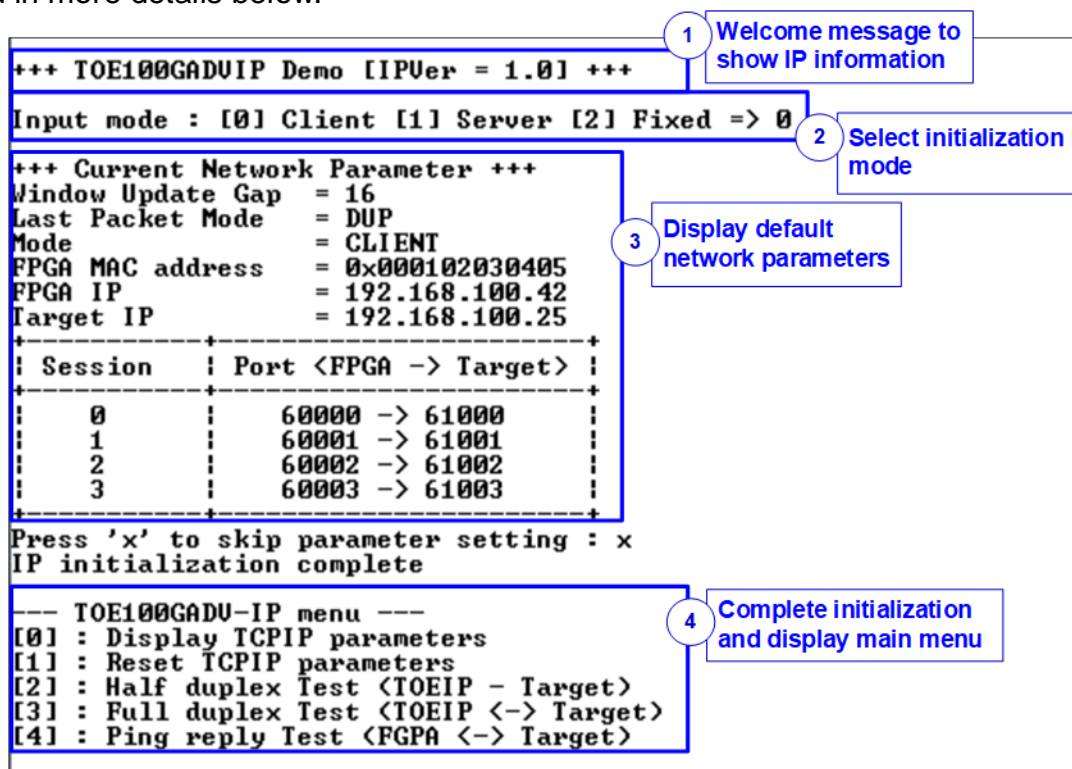


Figure 3-1 System initialization in Client mode by using default parameters

Figure 3-1 illustrates the four-step hardware initialization process, 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 TOE100GADV-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 TOE100GADV-IP. Details for each initialization mode are outlined as follows.
 - a) *Client mode:* TOE100GADV-IP sends an ARP request packet to retrieve the MAC address of the target device from the ARP reply packet.
 - b) *Server mode:* TOE100GADV-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 TOE100GADV-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 TOE100GADV-IP, while target parameters are the parameters of a PC or another FPGA.

3.2 Reset parameters

This menu facilitates to modify TOE100GADV-IP parameters, including the initialization mode, IP address, and source port number. As each parameter undergoes an update to be TOE100GADV-IP input signals, the CPU asserts reset signals to both TOE100GADV-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 TOE100GADV-IP, UserDataGen, and UserDataVer by setting HW_RST_INTREG[0] to 1b.

- 5) Set all parameters of TOE100GADV-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, TOE100GADV-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 logic to showcase optimal performance, disabling the data verification function 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 TOE100GADV-IP (TOE_TCS0LH_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_LENL/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_INT_REG) 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_INT_REG) 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.

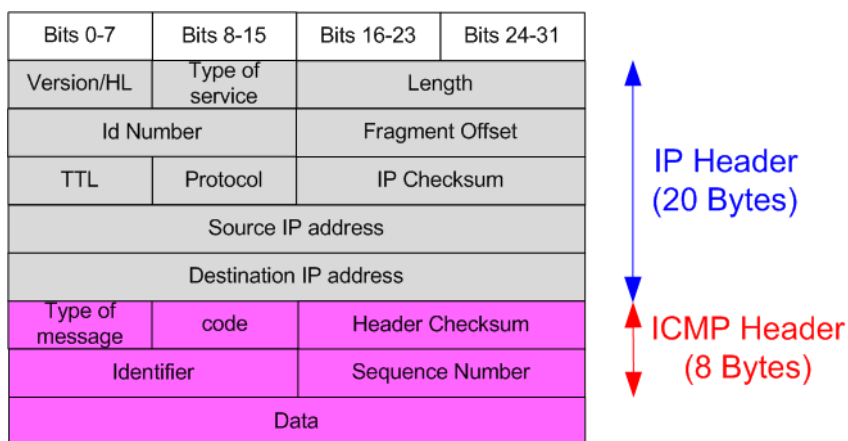


Figure 3-2 Packet structure for ICMP request/reply packet

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))

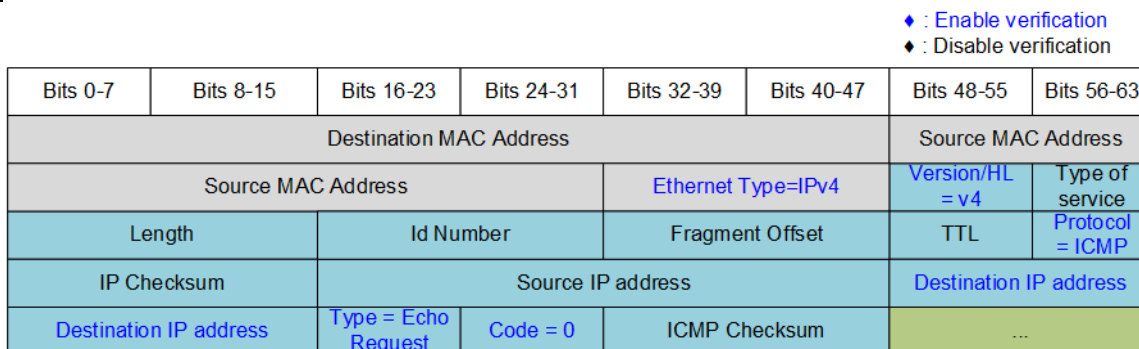


Figure 3-3 ICMP Echo request packet filtering

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 bytes) = 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[5:0]). 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 Functions 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' parameters 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/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 detect 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.

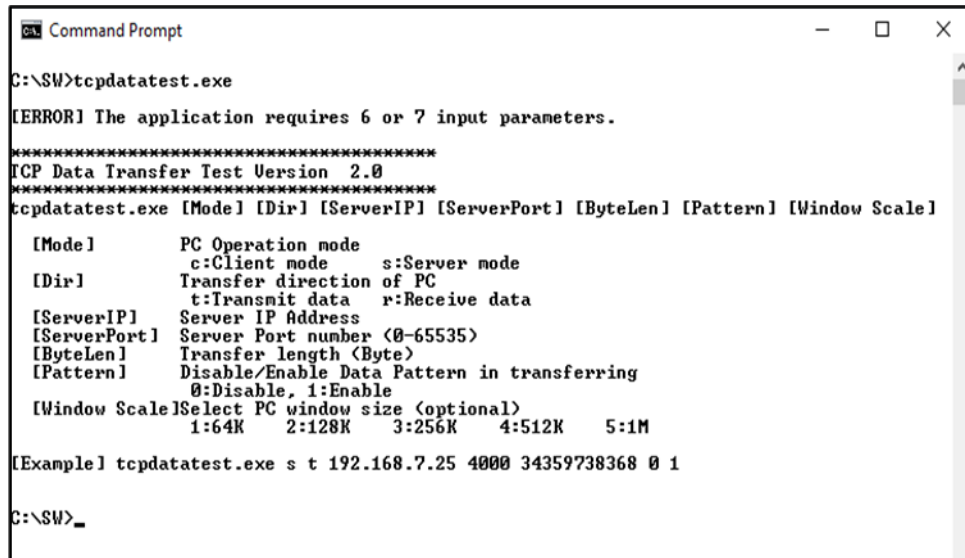
void init_filter (void)	
Parameters	None
Return value	None
Description	Disable UserRxEMAC by setting RXEMAC_CMD_INTREG to block the received packet. Next, Write RXEMAC_HDVAL_ADDR and RXEMAC_HDEN_ADDR for filtering an ICMP Echo request packet, as shown in the blue text of Figure 3-3.

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



```

Command Prompt
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 192.168.100.42)
- 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 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 is 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 the 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 steps 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

```

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_single” application usage

The “tcp_client_txrx_single” application allows the PC to send and receive TCP data through Ethernet using the same port number simultaneously. It operates exclusively in Client mode and necessitates the input of server parameters (network parameters set to this hardware) 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 to operate, 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 the 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 the 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
1.0	9-Feb-24	Initial version release