NVMeTCP25G-IP on Alveo card reference design

Rev1.0 27-Mar-23

1 Introduction

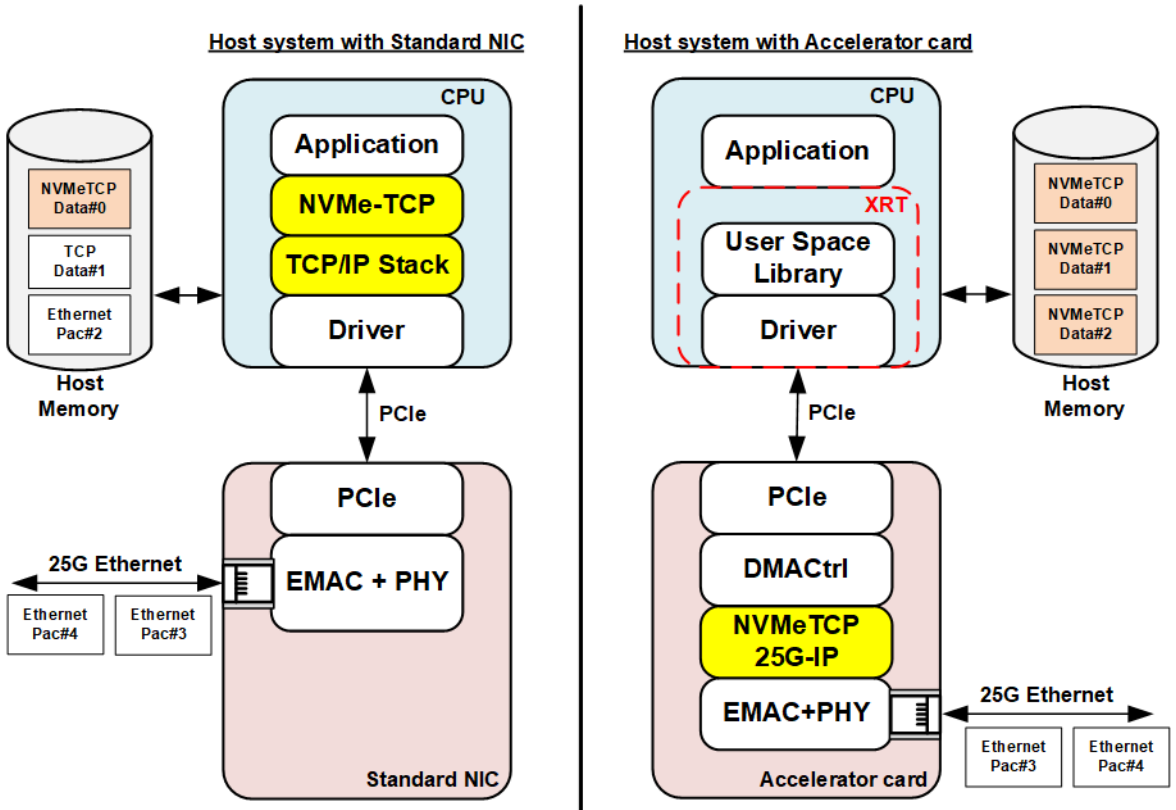

Figure 1-1 Standard NIC and Accelerator card comparison

The NVMe/TCP protocol is a type of NVMe-oF that enables remote hosts to access NVMe SSD on the target system via the network. Standard NICs are used to transfer command requests and data between hosts and targets over the network. The NVMe/TCP system's standard driver implements the TCP/IP Stack and NVMe-TCP protocol on both the host and the target. As a result, this requires system resources for various functions, memory access, and memory usages for storing several data types (Ethernet packet for EMAC function, TCP payload data for TCP Stack function, and NVMeTCP data for NVMe-TCP function, as shown in the left side of Figure 1-1). The host system resource is the bottle-neck to achieve the high-performance data transfer.

To reduce these resource requirements, the NVMeTCP25G-IP is a fully offload engine that features the TCP/IP stack and NVMe-TCP function for the host's write/read operations on the NVMe SSD at the target system across 25G Ethernet network. In this reference design, the NVMeTCP25G-IP is integrated into the Alveo accelerator card, which is a hardware platform from Xilinx that provides both hardware and software platforms PCIe communication. The software platform, called XRT, implements the functions for the host to access hardware registers and enables hardware to directly access the Host memory at a high-speed rate.

By using the NVMeTCP25G-IP on the Alveo accelerator card, the host system can perform NVMe/TCP host function without using CPU and memory resource for processing TCP/IP Stack and NVMe-TCP function. The Host memory only stores NVMeTCP data, which is the user data of the NVMeTCP25G-IP, as shown in the right side of Figure 1-1. Using the high-performance host memory access provided by XRT, the host system can write and read the data from the target system at high-speed rate across the 25G Ethernet network.

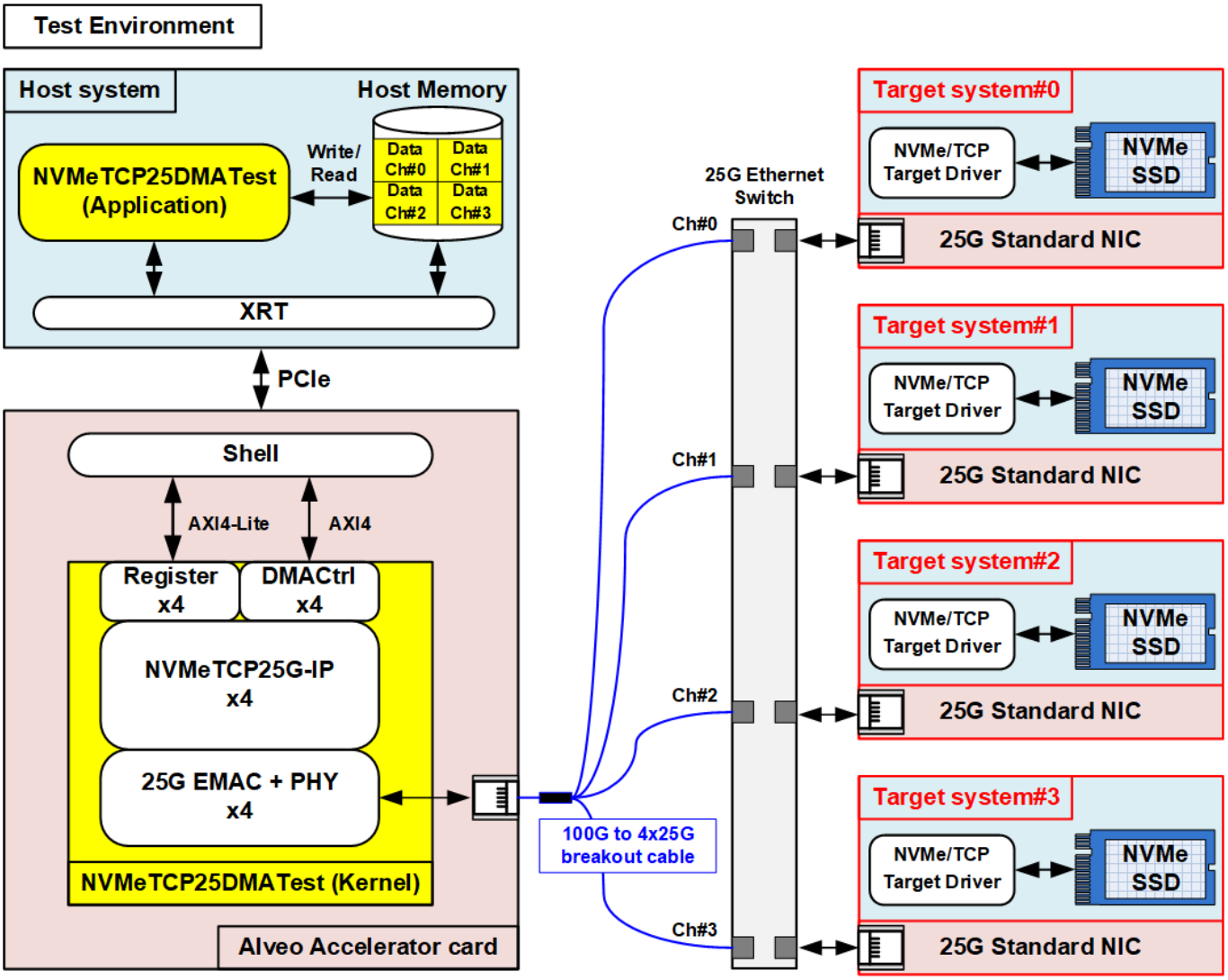

Figure 1-2 Test environment of NVMeTCP25G-IP on Alveo card.

The Alveo Accelerator card has the capability to use a 100G Ethernet connection via a QSFP28 connector, allowing it to be connected to four 25G Ethernet MAC and PHY modules for communication with four target systems through a 25G Ethernet network. In this reference design, four 25G Ethernet connections are utilized to implement four NVMe/TCP hosts, as shown in Figure 1-2.

The NVMeTCP25DMATest logic consists of four hardware sets comprising 25GEMAC and PHY, NVMeTCP25G-IP, DMA controller (DMACtrl), and Register modules. Each DMACtrl transfers user data of each NVMeTCP25G-IP to the Host memory via AXI4 bus independently and the user data of each NVMeTCP25G-IP is stored separately. DMACtrl and NVMeTCP25G-IP parameters and status signals of each connection are stored in the Register module, which is connected to AXI4-Lite bus to enable the CPU to write or read the hardware registers.

The NVMeTCP25DMATest application is a test application that runs on the Host system and handles user data of four NVmeTCP25G-IP modules stored in the Host memory simultaneously. The application writes or verifies test data on the Host memory and measures the test performance of each channel to provide the user with a test result. It also sets the hardware registers to initiate the Write or Read command and then reads the hardware registers to monitor the test status when the operation is not completed. As a result, the application requires the Host system resource for handling data on the Host memory and monitoring progress by polling method. However, if the user's application does not handle large data on the Host memory and uses the interrupt to handle the hardware, it will require less Host system resource to achieve good Write/Read performance.

For more information on preparing the host system for using the Alveo accelerator card, please visit the Xilinx website.
https://www.xilinx.com/products/boards-and-kits/alveo.html

Design Gateway also provides the host system for the Alveo card, Turnkey Accelerator System, which can be checked in more detail on our website.
https://dgway.com/AcceleratorCards.html

In the document, topic 2 shows the details of the hardware design on the Alveo card and topic 3 describes software implementation on the Accelerator system.

2 NVMeTCP25DMATest (Hardware)

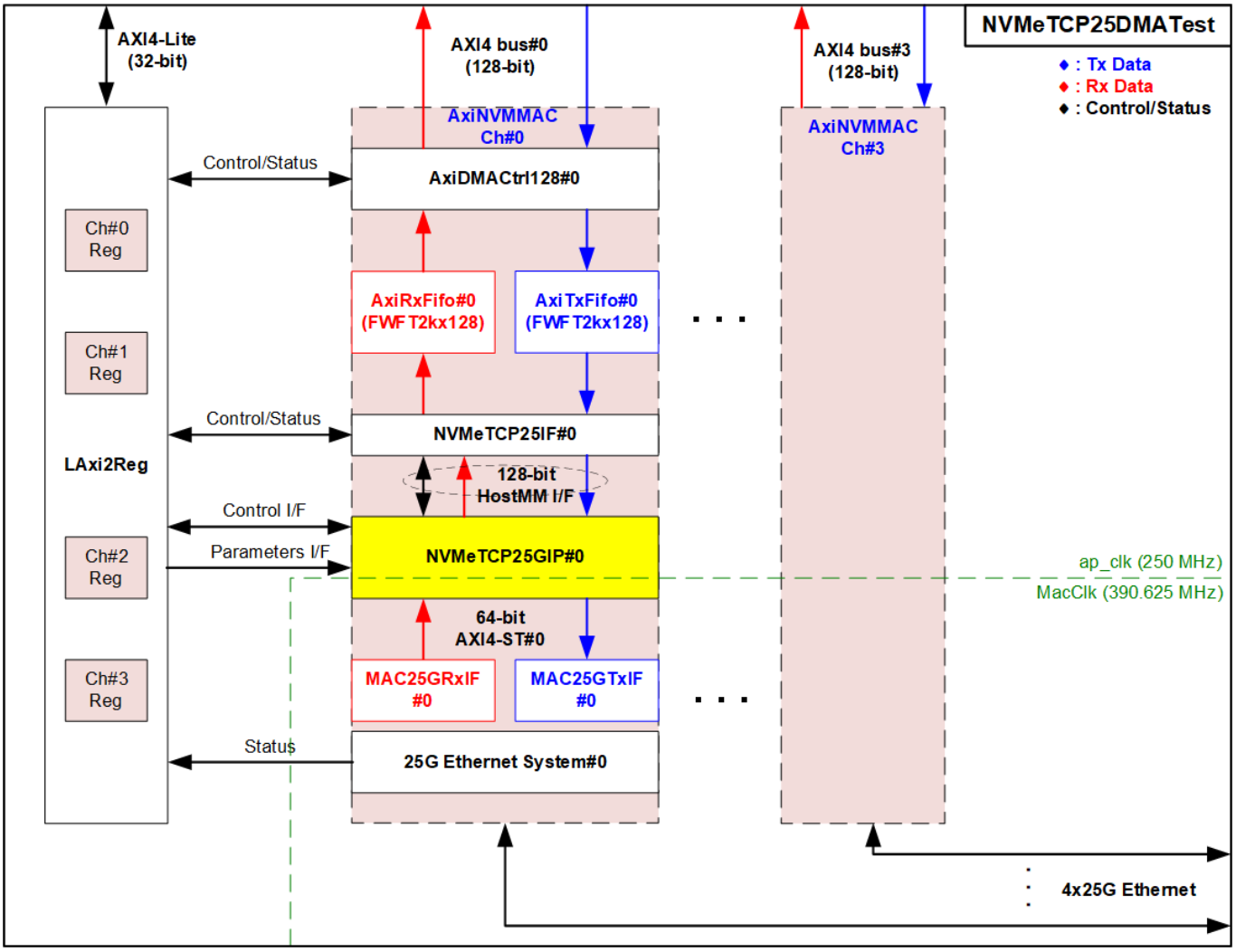

Figure 2-1 NVMeTCP25DMATest block diagram

The NVMeTCP25DMATest hardware kernel has two interfaces for connecting to the Host system: a 32-bit AXI4-Lite bus for hardware register access and four 128-bit AXI4 buses for data transfer with Host memory. The kernel handles four 25G Ethernet connections, so it has four sets of AxiNVMMAC and Register to operate independently.

To enable 25G network communication, the NVMeTCP25G-IP core engine within AxiNVMMAC requires a connection to the 25G Ethernet system. This can be achieved using different approaches, such as the Ethernet MAC core provided by Design Gateway and Xilinx 10G/25G Ethernet subsystem. With Design Gateway's Ethernet MAC core, NVMeTCP25G-IP can be directly connected via a 64-bit AXI4-Stream bus. However, with Xilinx 10G/25G Ethernet subsystem, adapter logics, MAC25GRxIF and MAC25GTxIF, are needed to buffer the data stream transferred between the Ethernet MAC and NVMeTCP25G-IP.

The HostMM I/F of NVMeTCP25G-IP for sending Write/Read command and transferring data is connected to NVMeTCP25IF, which is the interface logic for streaming the Write/Read data between the AxiFIFOs and NVMeTCP25G-IP. AxiFIFOs are connected to AxiDMACtrl, which is a DMA engine for data transfer with the Host memory via AXI4 bus. The Parameters I/F for parameter assignment and Control I/F for control/status signal of NVMeTCP25G-IP are connected to LAxi2Reg for Host system write and read access. Similarly, the control and status of other modules are mapped to LAxi2Reg for Host system control and monitoring.

The NVMeTCP25DMATest uses two clock domains: MacClk for 25G Ethernet system operation and ap_clk for the application operation. The MacClk is fixed at 390.625 MHz for 25G Ethernet connection, which the ap_clk is set to 250 MHz for high-performance operation. The ap_clk is the application clock which can be configured by Vitis tool. The logic for clock-crossing domain (CDC) operation is implemented inside NVMeTCP25G-IP. According to NVMeTCP25G-IP datasheet, the clock frequency of user interface (ap_clk) must be greater than or equal to 195.3125 MHz.

More details of each hardware module inside the NVMeTCP25DMATest are described as follows.

2.1 25G Ethernet System (25G BASE-SR)

25G Ethernet system consists of the MAC layer and PCS/PMA layer, and it can enable the RS-FEC feature to correct error found on the Ethernet link. The Ethernet MAC user interface is 64-bit AXI4-stream interface running at 390.625 MHz, and the Physical layer uses the 25G BASE-SR connection. Various solutions can be used to implement a 25G Ethernet system, and two solutions are explained in this document.

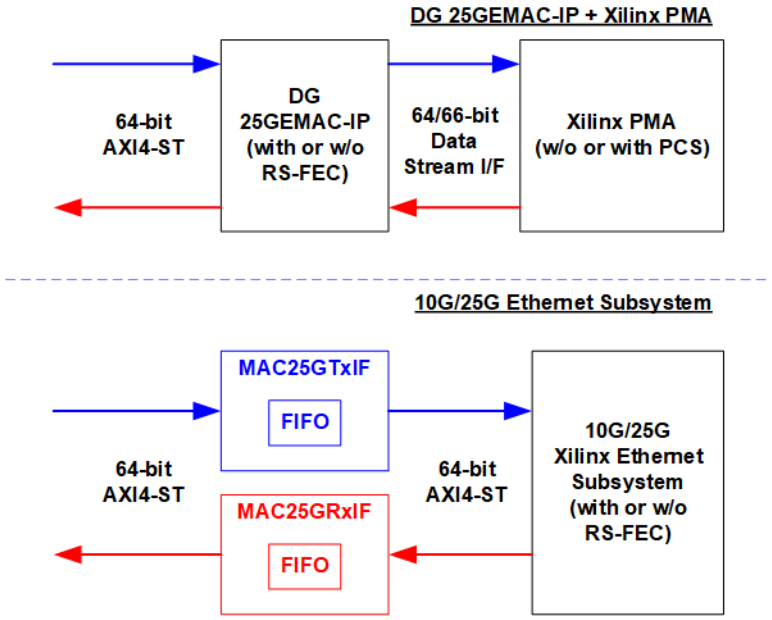

Figure 2-2 25G Ethernet System

Design Gateway offers the first solution, which provides two options: enable or disable the RS-FEC feature. To enable the RS-FEC feature, the 25GEMAC/PCS+RS-FEC IP with Xilinx PMA module is used to connecting to NVMeTCP25G-IP via a 64-bit AXI4-Stream directly. Without the RS-FEC feature, the 10G25GEMAC-IP with Xilinx PCS/PMA module is used to connect directly with NVMeTCP25G-IP. More information about the Ethernet MAC solution by Design Gateway can be found on our website.
https://dgway.com/NVMeTCP-IP_X_E.html

The second solution is provided by Xilinx, which offers the 10G/25G Ethernet Subsystem. The IP wizard allows the user to enable or disable the RS-FEC feature, and the EMAC, PCS, and PMA logic are included in this IP core. However, the user interface of the Xilinx IP, which is 64-bit AXI4 stream, is not compliant with the EMAC interface of the NVMeTCP25G-IP. The Xilinx IP may pause data transmission while transferring a packet, but NVMeTCP25G-IP requires continuous data transfer for each packet. Therefore, two adapter logics – MAC25GTxIF and MAC25GRxIF - must be designed to buffer the data when the Xilinx IP pauses data transmission. More information about Xilinx Ethernet MAC can be found on their website.
https://www.xilinx.com/products/intellectual-property/ef-di-25gemac.html

Note: This design enables RS-FEC feature for 25G Ethernet system, so please confirm that the network equipment can support RS-FEC before running this demo.

The details of the adapter logics, MAC25GTxIF and MAC25GRxIF, are described as follows.

MAC25GTxIF

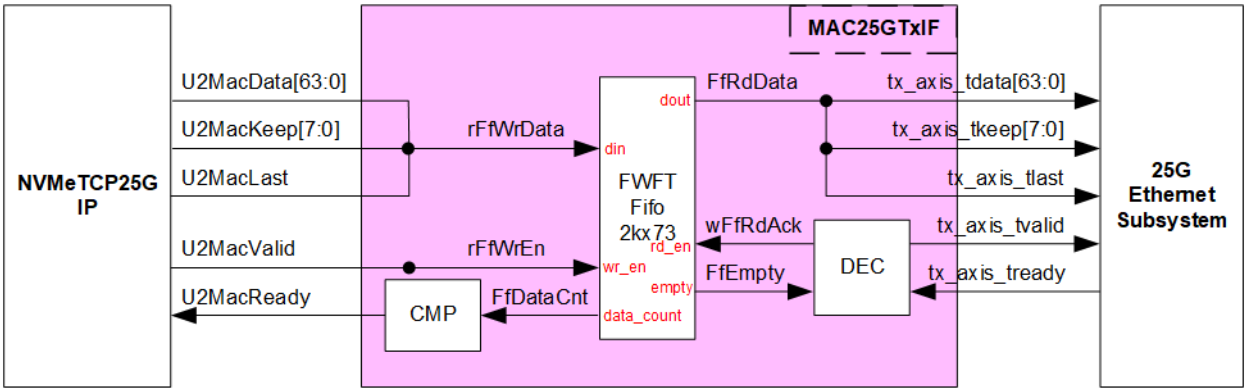

Figure 2-3 MAC25GTxIF Block diagram

According to NVMeTCP25G-IP specification, each packet must be continuously sent to the Ethernet MAC, so U2MacReady should remain asserted until the end of the packet. However, the Xilinx Ethernet Subsystem may pause data transmission by de-asserting tx_axis_tready while a packet is transmitting.

To solve this issue, the MAC25GTxIF is designed to store transmitted data from NVMeTCP25G-IP when EMAC is not ready to receive the new data. The FIFO depth of 2048 is sufficient to store at least one data packet during the pausing time. The maximum transmitted packet size is 8960 bytes or 1120 of 64-bit data, which fits within the FIFO/s storage capacity.

The FIFO is First-Word Fall-Through (FWFT) FIFO, which means that the read data is valid for reading at the same time as asserting read enable to 1b. Once the read enable of FIFO is asserted to 1b, the next read data is available on the read data bus in the following clock cycle.

The logic of MAC25GTxIF is split into two groups: Writing FIFO and Reading FIFO. The timing diagrams for each group are displayed in Figure 2-4 and Figure 2-5, respectively.

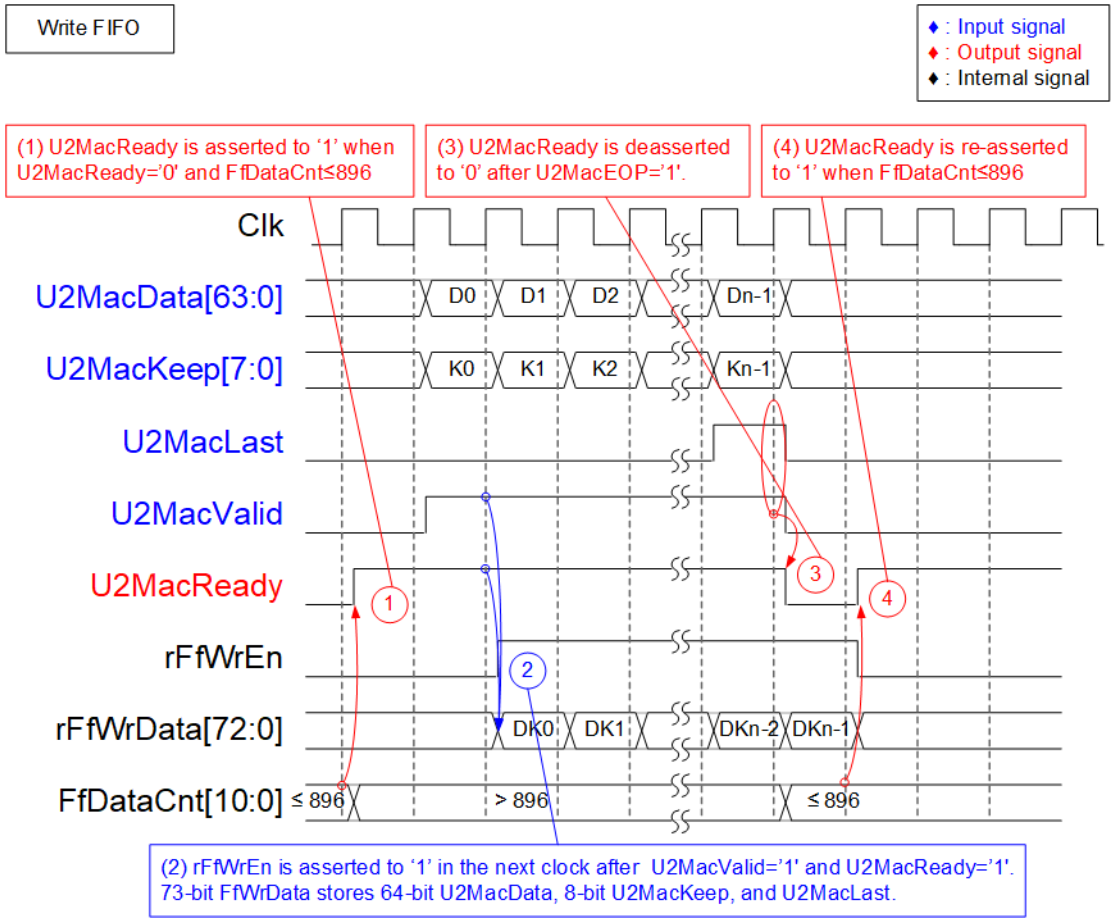

Figure 2-4 Timing diagram for transferring data from NVMeTCP25G-IP to FIFO

1) Before asserting U2MacReady to 1b for receiving a new packet from user, two conditions must be satisfied. Firstly, The FIFO must have sufficient free space to store the maximum packet size of 9014 bytes. A simple monitoring logic can be implemented by checking the upper bit of FfDataCnt to ensure that the data in the FIFO does not exceed 896 data, leaving 1151 data of free space. Secondly, the previous packet must have been completely transferred, as indicated by U2MacReady being set to 0b.
2) The user initiates packet transmission by setting U2MacValid to 1b. The input signals from user (U2MacData, U2MacKeep, and U2MacLast) are valid and stored in FIFO once U2MacValid and U2MacReady are both set to 1b. Subsequently, the inputs are written to the FIFO by setting rFfWrEn to 1b. The 73-bit write data to FIFO comprises of 64-bit data (U2MacData), 8-bit empty byte (U2MacKeep), and end flag (U2MacLast).
3) After receiving the final data of a packet (U2MacLast=1b and U2MacValid=1b), U2MacReady is de-asserted to 0b to pause data transmission to allow reading of FfDataCnt.
4) If FfDataCnt indicates that the FIFO has enough free space, U2MacReady will be re-asserted to 1b in the next cycle.

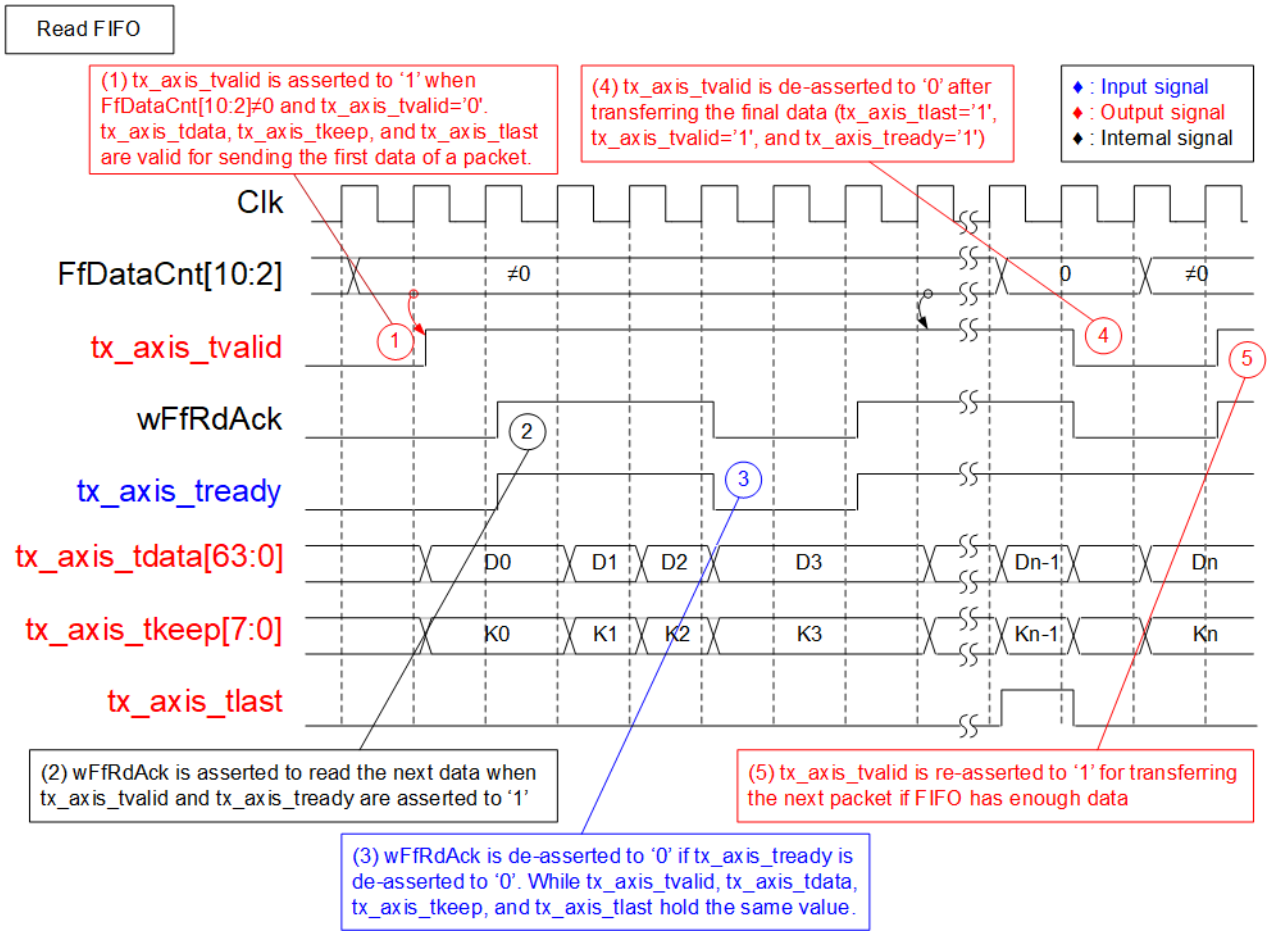

Figure 2-5 Timing diagram for transferring data from FIFO to EMAC

1) Data transmission of a new packet begins when the FIFO has stored some data (FfDataCnt[10:2] ≠ 0) and the packet is not transmitting (tx_axis_tvalid=0b). To initiate data transmission, tx_axis_tvalid is asserted to 1b along with valid output signals to EMAC, i.e., 64-bit tx_axis_tdata, 8-bit tx_axis_tkeep, and tx_axis_tlast.
2) Once the data is transmitted to EMAC completely (tx_axis_tvalid =1b and tx_axis_tready=1b), wFfRdAck is asserted to 1b to read the next data from the FIFO.
3) If tx_axis_tready is de-asserted to 0b, wFfRdAck will also be de-asserted to 0b, which pauses the reading of new data from the FIFO. Therefore, all output signals sent to EMAC will hold the same value until EMAC re-asserts tx_axis_tready to 1'.
4) After the final data of a packet has been completely transferred (tx_axis_tlast=1b and tx_axis_tready=1b), tx_axis_tvalid is de-asserted to 0b, which pauses data transmission and allows for checking the data size in FIFO to transfer the next packet.
5) Transmission of the next packet occurs only when the FIFO has sufficient data, and the process returns to step 1 to transmit the new packet.

MAC25GRxIF

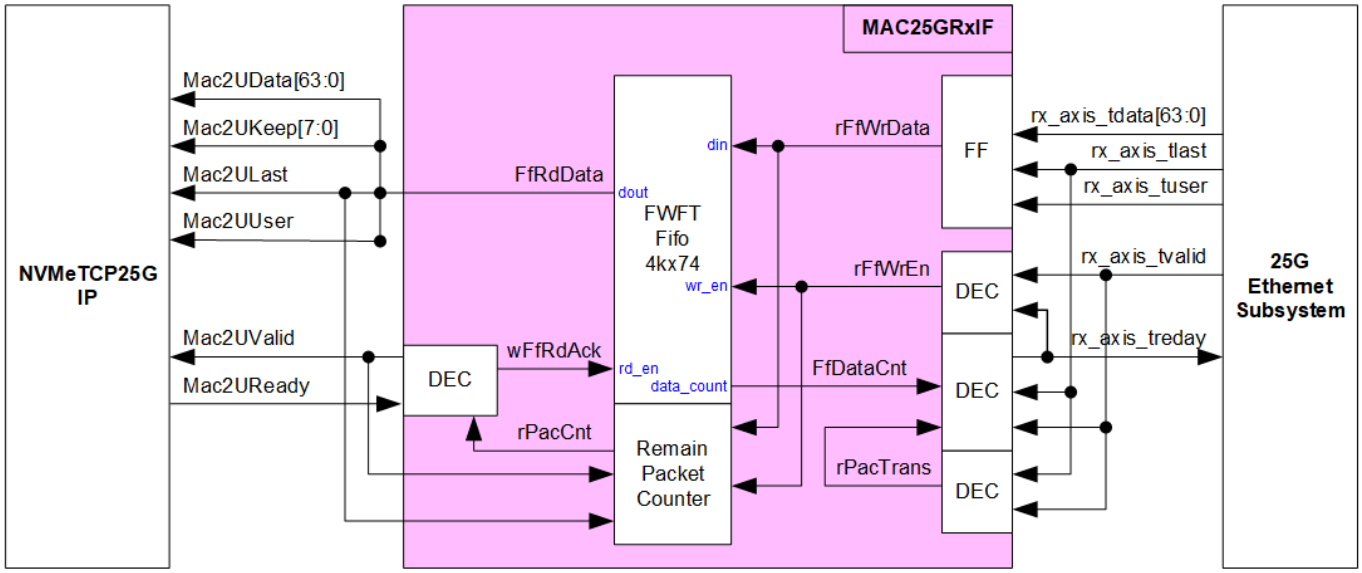

Figure 2-6 MAC25GRxIF Block diagram

The EMAC interface of NVMeTCP25G-IP requires to receive the packets continuously, so Mac2UValid must remain set to 1b from the beginning to the end of each packet. However, the Xilinx Ethernet Subsystem may pause data transmission by de-asserting the valid signal (rx_axis_tvalid) before the packet is complete.

To solve this issue, the MAC25GRxIF is designed to store all packet data transmitted from EMAC before forwarding it to NVMeTCP25G-IP. This guarantees that all data is available for transfer until the end of the packet. The FIFO has a depth of 4096, which can store multiple Ethernet packets, and operates as a First-Word Fall-Through FIFO.

The Remain packet counter keeps track of the number of packets stored in the FIFO. When a new packet is received from EMAC, the counter is increased, and it decreased when the packet is completely forwarded to NVMeTCP25G-IP.

The logic of MAC25GRxIF is split into two groups: Writing FIFO and Reading FIFO. The timing diagrams for each group are displayed in Figure 2-7 and Figure 2-8, respectively.

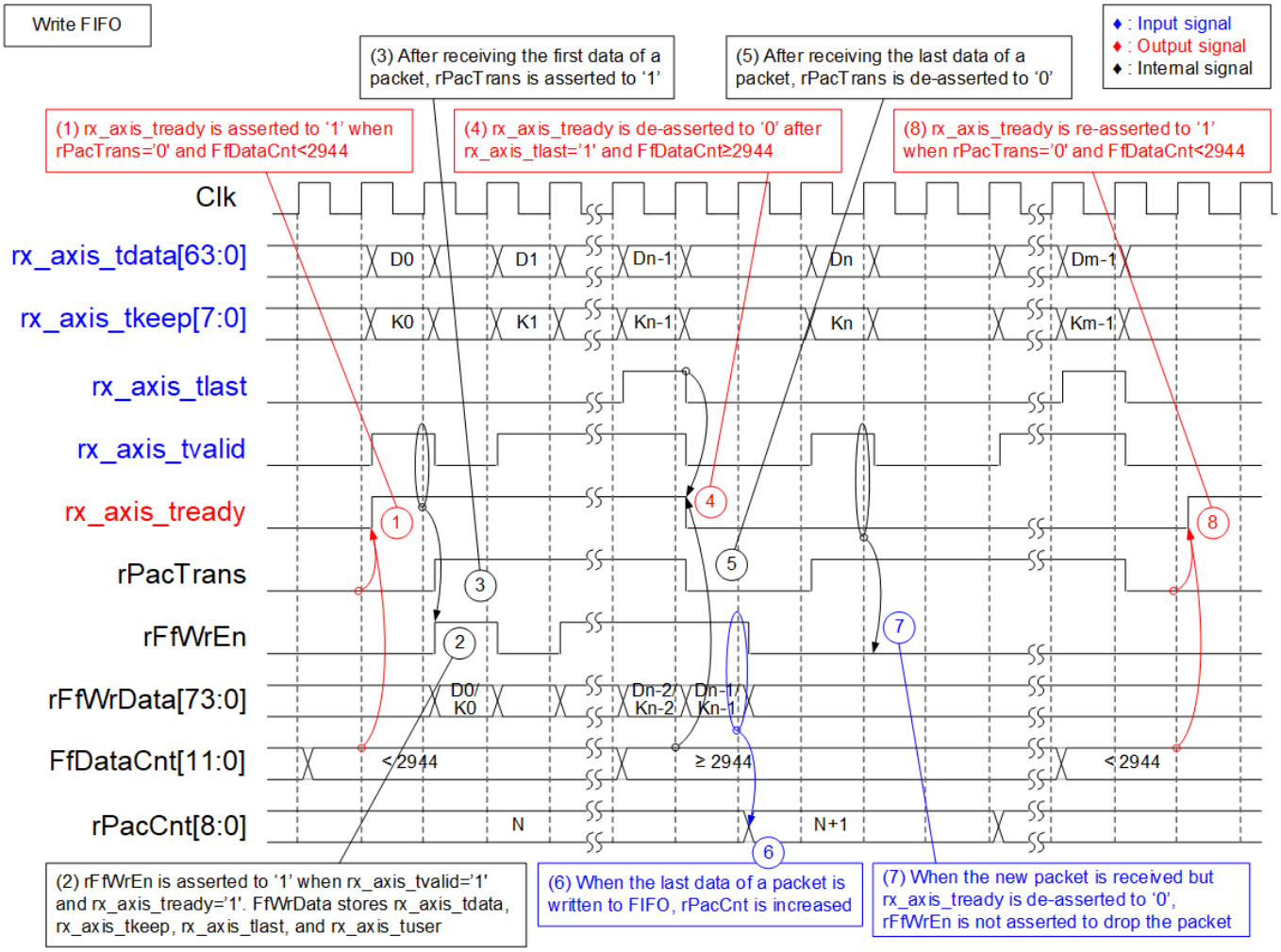

Figure 2-7 Timing diagram for transferring data from EMAC to FIFO

1) The free space size in FIFO is checked by reading FfDataCnt. If it is less than 2944 (the free space in FIFO is more than 1153), this is sufficient to store the maximum packet size (9014 bytes). Also, rPacTrans must be equal to 0 to confirm that the packet is not currently being transmitted. Once these conditions are met, rx_axis_tready is asserted to 1b to begin receiving data from EMAC.
2) When a new packet is ready to be transferred (rx_axis_tvalid is set to 1b), the data and control signals from EMAC are stored in the FIFO, including 64-bit rx_axis_tdata, 8-bit rx_axis_tkeep, rx_axis_tlast, and rx_axis_tuser. rFfWrEn is set to 1b to write the 74-bit data to the FIFO.
3) After the first data of a packet is received, rPacTrans is set to 1b and remains until the end of packet is received. This allows to use rPacTrans for monitoring the packet transmission status.
4) If the final data of a packet is received and there is not enough free space in the FIFO (FFDataCnt≥2944), rx_axis_tready is de-asserted to 0b to pause data reception.
5) After the final data of a packet is received, rPacTrans is de-asserted to 0b to change the packet transmission status from Busy to Idle.
6) Once the final data of a packet has been stored in the FIFO (rFfWrEn=1b and rFfWrData[72] which is last flag =1b), the packet counter (rPacCnt) is increased by 1 to indicate the total number of packets stored in the FIFO.

7) If a new packet is received but rx_axis_tready is still de-asserted to 0b, the received packet will be dropped and not be stored in the FIFO.
8) When there is no packet currently being transmitted and there is enough free space in the FIFO, rx_axis_tready is re-asserted to 1b to resume data reception.

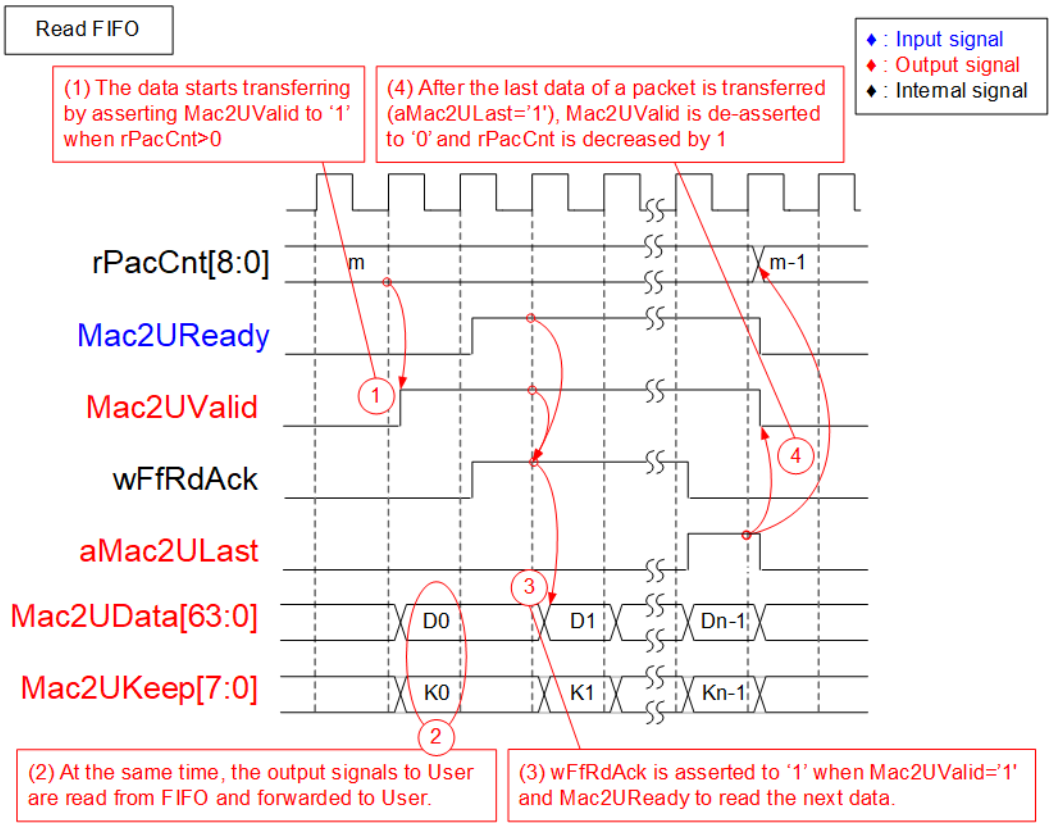

Figure 2-8 Timing diagram of data transferring from FIFO to NVMeTCP25G-IP

1) Data transmission from FIFO to NVMeTCP25G-IP begins only when there is at least one packet stored in FIFO (indicated by rPacCnt not being equal to 0). To start the transmission, Mac2UValid is asserted to 1b.
2) The data and control signals read from FIFO are sent to NVMeTCP25G-IP, including 64-bit Mac2UData, 8-bit Mac2UKeep, Mac2ULast, and Mac2UUser. Mac2UValid remains asserted to 1b until the end of packet to ensure continuous data transmission.
3) Once the data is completely transferred to NVMeTCP25G-IP (Mac2UValid=1b and Mac2UReady=1b), wFfRdAck is asserted to 1b to read the next data.
4) After the final data of a packet is transferred (aMac2ULast=1b and Mac2UValid=1b), Mac2UValid and wFfRdAck are de-asserted to 0b to pause a packet transmission. Also, rPacCnt is decreased by 1 after the completion of one packet transfer.

2.2 NVMeTCP-IP for 25G

The NVMeTCP25G-IP is the product that enables NVMe over TCP, making it possible to access a target SSD via 25G Ethernet using a TCP/IP stack and host controller offload engine. With support for both Write and Read commands at high-speed rate, the NVMeTCP25G-IP is a solution for data transfer across the network without using the CPU and external memory. To get started with data transfer, simply set the target name and execute the Connect operation on the NVMeTCP25G-IP. To end the connection between the IP and the target, execute the Disconnection operation. The user interface of the IP is divided into three groups - Parameters, Control, and Memory Map (HostMM) Interface. For more detailed information about this IP product, please visit our website. https://dgway.com/products/IP/NVMeTCP-IP/dg_nvmetcp25g_ip_data_sheet_xilinx.pdf

2.3 NVMeTCP25IF

The following section provides an overview of NVMeTCP25IF, a logic designed to facilitate data transfer between AxiTxFifo/AxiRxFifo and NVMeTCP25G-IP. While the demo system allows users to input start address and transfer size for writing or reading data from the target SSD across 25G Ethernet, NVMeTCP25G-IP has a fixed data size per Write/Read command of 8 Kbytes. To process larger transfer sizes, NVNeTCP25IF generates multiple Write/Read commands to NVMeTCP25G-IP. This section describes the details of NVMTCP25IF design.

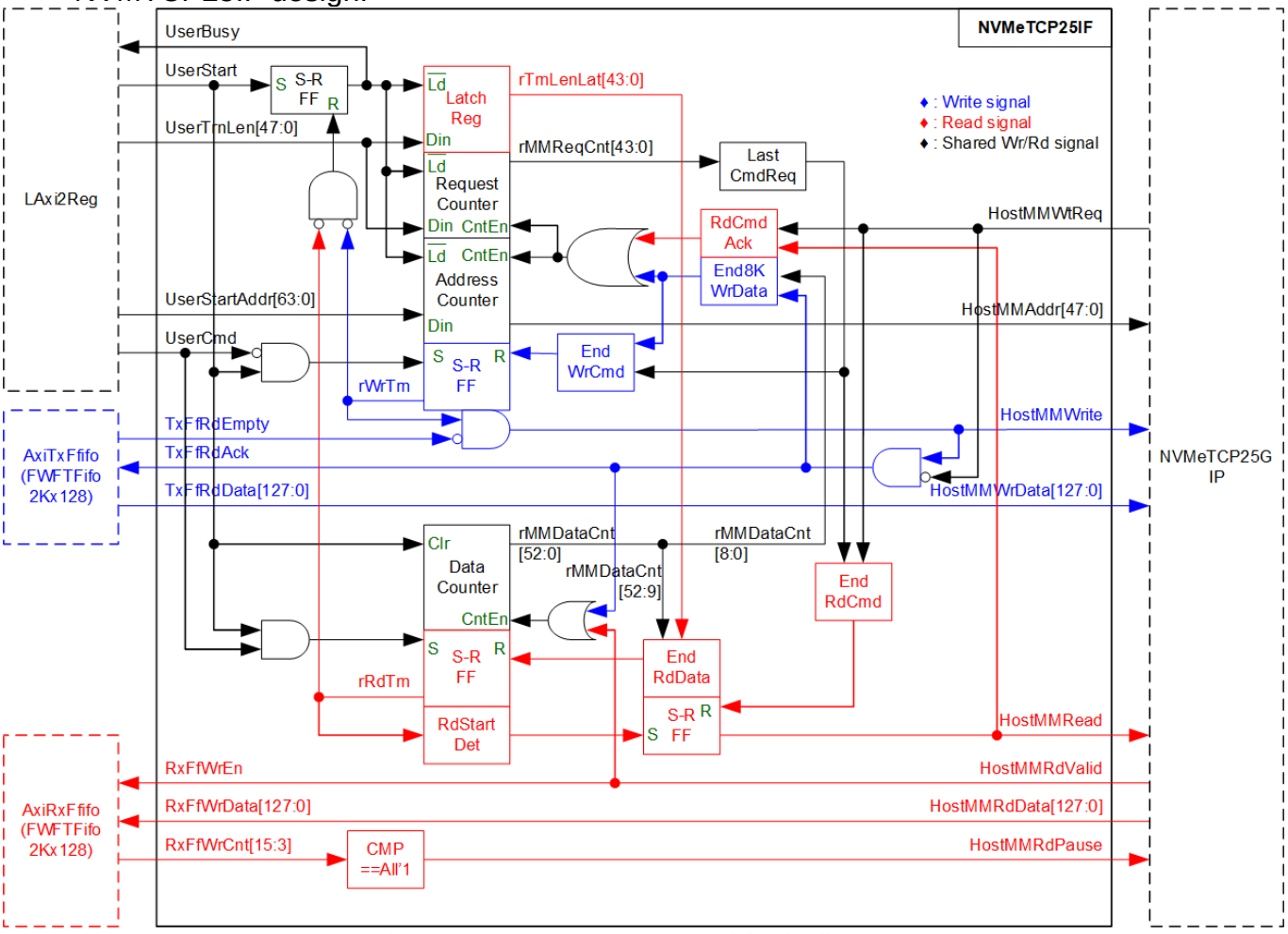

Figure 2-9 NVMeTCP25IF block diagram

To start a write/read transfer, the user must set UserStart to 1b and provide three parameters - UserCmd (Write or Read), UserStartAddr (the starting address to write/read in 512-byte unit), and UserTrnLen (the total transfer length in 512-byte unit). Once set, UserBusy is set to 1b to indicate that the command is processing. UserCmd is decoded to assert either rWrTrn (for Write command) or rRdTrn (for Read command) to specify the command type. Once the operation is completed, rWrTrn/rRdTrn and UserBusy are de-asserted.

For a Write command, when NVMeTCP25G-IP has free space to receive the data (HostMMWtReq=0b) and AxiTxFifo has data for transfer (TxFfRdEmpty=0b), data is sent from AxiTxFifo (TxFfRdData) to NVMeTCP25G-IP (HostMMWrData) with asserting HostMMWrite to 1b. HostMMWrite is used for sending both 8KB Write command request and 8KB Write data (the command size per each Write command) to NVMeTCP25G-IP. HostMMWtReq is used by NVMeTCP25G-IP to indicate that the Write command and the Write data is accepted.

For a Read command, HostMMRead is asserted to send 8KB Read command request and NVMeTCP25G-IP de-asserts HostMMWtReq to 0b to indicate that the 8KB Read command is accepted. Following this, 8KB data is returned from NVMeTCP25G-IP by asserting HostMMRdValid to 1b. HostMMRdData is then stored in AxiRxFifo via RxFfWrData signal. When AxiRxFifo is almost full, indicated by RxFfWrCnt, HostMMRdPause is asserted to 1b to request NVMeTCP25GIP to pause data transmission. HostMMRdValid is then de-asserted to 0b.

UserStartAddr is loaded into the Address counter to generate the HostMMAddr for each 8KB Write/Read command sent to NVMeTCP25G-IP. The counter is incremented to the next 8KB address by two different conditions, depending on the command type. For a Write command, the counter is incremented after the last data of the 8KB block is completely transferred to NVMeTCP25G-IP, indicated by TxFfRdAck (data accepted) and rMMDataCnt (the amount of transferred data). For a Read command, the Address counter is incremented after NVMeTCP25G-IP accepts the 8KB Read command request, indicated by HostMMWtReq and HostMMRead.

UserTrnLen is loaded into two logic blocks, the Latch register of rTrnLenLat and the Request Counter. rTrnLenLat is used to check the end of data transfer for Read command. When the total number of Read data (rMMDataCnt) is equal to this value, it indicates that the Read command is completed, and rRdTrn is set to 0b. The Request Counter is a down-counter used to check the remaining 8KB command request counts that have not been sent to NVMeTCP25G-IP. The Request Counter is decremented under the same conditions that the Address Counter is incremented. The Request Counter output, rMMReqCnt, can detect when the last 8KB command request has been completely sent to NVMeTCP25G-IP. This is a condition to de-assert rWrTrn and HostMMRead to 0b.

The Data Counter shows the amount of transferred data in both Write and Read commands. For a Write command, it is incremented when each data from AxiTxFifo is successfully transferred to NVMeTCP25G-IP (HostMMWrite=1b and HostMMWtReq=0b). For a Read command, it is incremented when the NVMeTCP25G-IP stores each data to AxiRxFifo by setting HostMMRdValid to 1b. The Data Counter output, rMMDataCnt, can detect when the last Write data of each 8KB command is transferred using only bit[8:0]. Similarly, it can also detect when the last Read data of the Read command is transferred using only bit[52:9].

Timing diagrams of NVMeTCP25IF while executing Write and Read command are shown in Figure 2-10 and Figure 2-11, respectively.

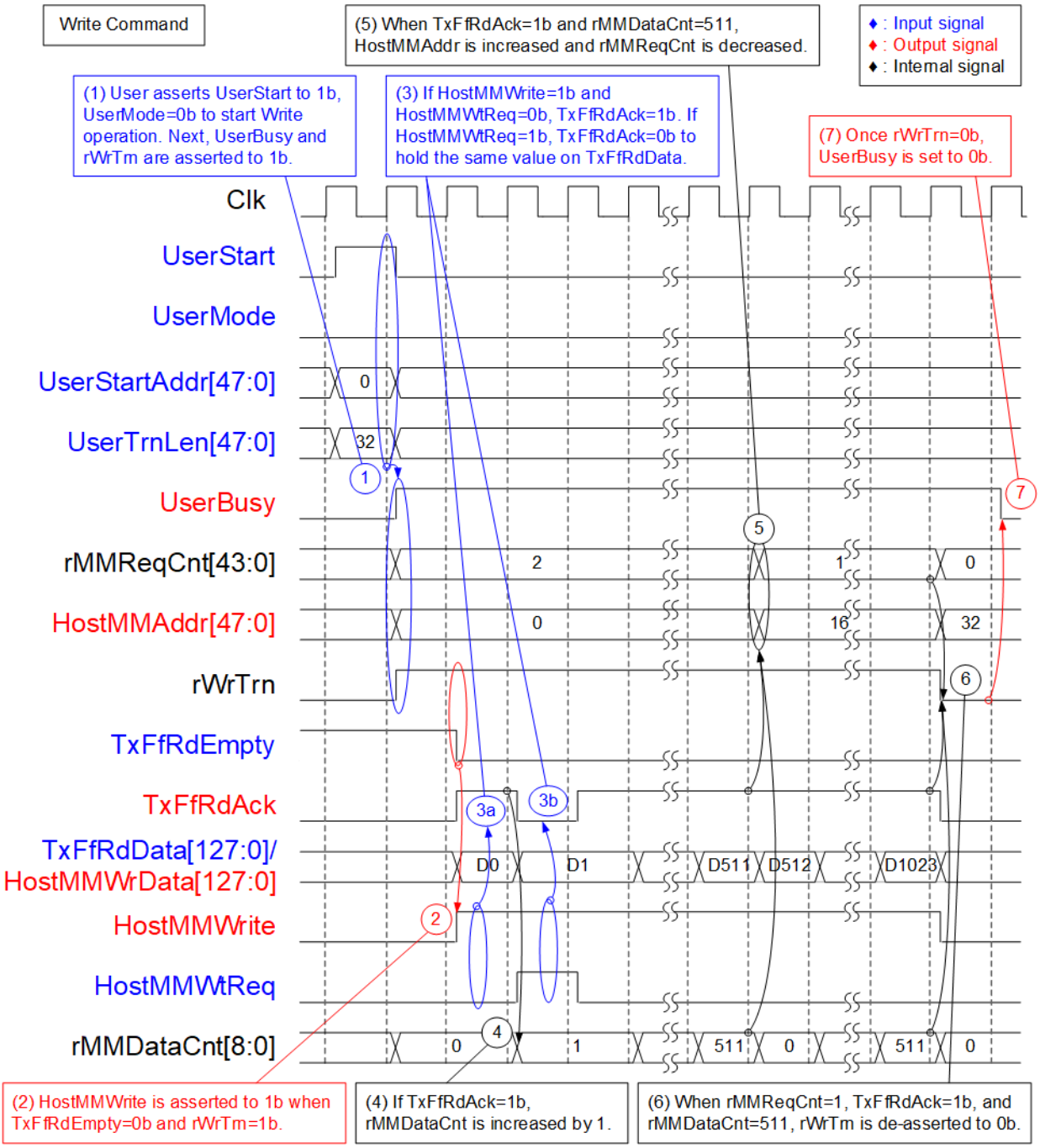

Figure 2-10 NVMeTCP25IF Write command timing diagram

Figure 2-10 shows timing diagram when the user sets 16KB Write command and NVMeTCP25IF generates two 8KB Write commands along with the data to the NVMeTCP25G-IP.

1) The Write command is initiated by the user by setting UserStart to 1b and UserMode to 0b. In this cycle, the start address (UserStartAddr) and total transfer length (UserTrnLen) are loaded, and the internal signals are initialized as follows.
   a) UserBusy and rWrTrn are set to 1b to indicate that the Write command is in progress.
   b) rMMReqCnt calculates the number of commands for sending to NVMeTCP25G-IP from UserTrnLen. For example, if UserTrnLen is 32 (16KB size), the result of rMMReqCnt is set to 2 (2 x 8KB command).
   c) HostMMAddr loads the value from UserStartAddr.
   d) rMMDataCnt is reset to zero.
2) If the Write command is still processing (rWrTrn=1b) and AxiTxFifo has data (TxFfRdEmpty= 0b), the Write command and Write data are transmitted to NVMeTCP25G-IP by setting HostMMWrite to 1b along with the valid HostMMWrData, which is directly mapped from the Read data of AxiTxFIFO (TxFfRdData).
3) When the NVMeTCP25G-IP accepts each transmitted data by de-asserting HostMMWtReq to 0b, the read enable of AxiTxFifo (TxFfRdAck) is set to 1b to read the next data from AxiTxFifo. If the NVMeTCP25G-IP de-asserts HostMMWtReq to 1b to pause data transmission, TxFfRdAck will be set to 0b to hold the same value on HostMMWrData.
4) After each data is accepted by the NVMeTCP25G-IP, indicated by TxFfRdAck=1b, the data counter (rMMDataCnt) is incremented.
5) Once the last data of each 8KB Write command is successfully transferred to NVMeTCP25G-IP (TxFfRdAck=1b and rMMDataCnt[8:0]=511), HostMMAddr is incremented to the next 8KB address and rMMReqCnt is decremented to update the remaining request count.
6) When the last data of the Write command is completely transferred to NVMeTCP25G-IP (TxFfRdAck=1b, rMMDataCnt[8:0]=511, and rMMReqCnt=1), rWrTrn is de-asserted to 0b to finish the Write operation.
7) Finally, UserBusy is set to 0b to inform the user that the operation is completed.

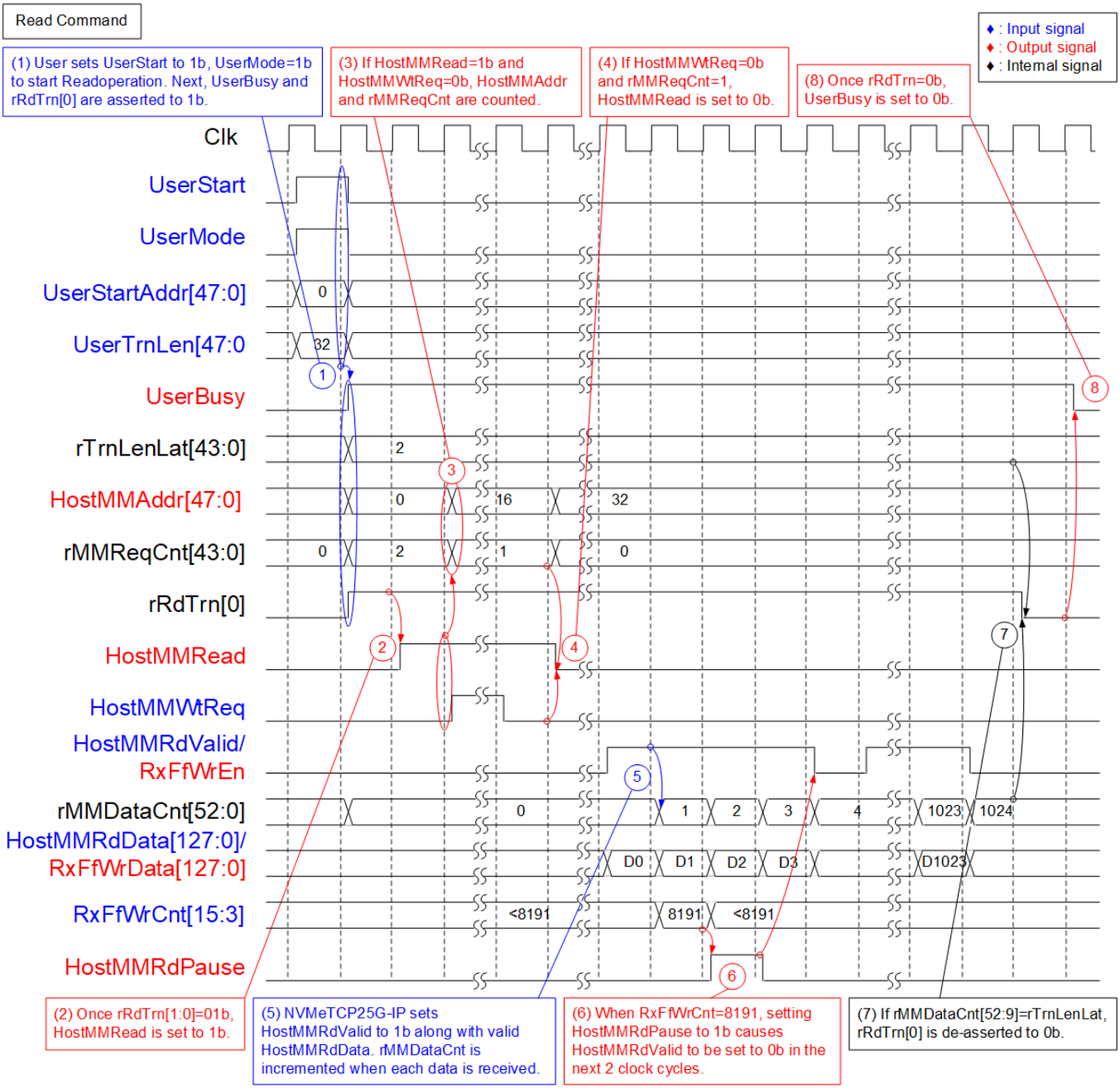

Figure 2-11 NVMeTCP25IF Read command timing diagram

Figure 2-11 shows timing diagram when the user sets 16KB Read command and then NVMeTCP25IF generates two 8KB Read commands to the NVMeTCP25G-IP. After that, two of 8KB Read data is returned from NVMeTCP25G-IP to write to AxiRxFifo.

1) The Read command is initiated by the user by setting UserStart to 1b and UserMode to 0b. In this cycle, the start address (UserStartAddr) and total transfer length (UserTrnLen) are loaded, and the internal signals are initialized as follows.
   a) UserBusy and rRdTrn are set to 1b to indicate that the Read command is in progress.
   b) rMMReqCnt and rTrnLenLat calculate the number of commands for sending to NVMeTCP25G-IP from UserTrnLen. For example, if UserTrnLen is 32 (16KB size), the results of rMMReqCnt and rTrnLenLat are set to 2 (2 x 8KB command).
   c) HostMMAddr loads start address from UserStartAddr.
   d) rMMDataCnt is reset to zero.
2) When the rising edge of rRdTrn[0] occurs, HostMMRead is set to 1b. HostMMRead holds its value until the last 8KB Read command is requested to NVMeTCP25G-IP.
3) Once the NVMeTCP25G-IP accepts each 8KB Read command request by de-asserting HostMMWtReq to 0b, HostMMAddr is incremented to the next 8KB address and rMMReqCnt is decremented to update the remaining request count.
4) When the last 8KB Read command request is accepted by NVMeTCP25G-IP (rMMReqCnt=1, HostMMRead=1b, and HostMMWtReq=0b), HostMMRead is set to 0b.
5) NVMeTCP25G-IP returns 8KB data of each Read command by setting HostMMRdValid to 1b along with the valid data on HostMMRdData, which are directly mapped to RxFfWrEn and RxFfWrData, respectively, for writing data to AxiRxFifo. The data counter (rMMDataCnt) is also incremented when HostMMRdValid is set to 1b.
6) If AxiRxFifo becomes almost full (RxFfWrCnt[15:3]=8191), HostMMRdPause is set to 1b to request NVMeTCP25G-IP to pause data transmission. As a result, NVMeTCP25G-IP de-asserts HostMMRdValid to 0b in the next two clock cycles.
7) Once the last data is received from NVMeTCP25G-IP, indicated by rMMDataCnt[52:9]= rTrnLenLat, rRdTrn[0] is de-asserted to 0b to finish the Read operation.
8) Finally, UserBusy is set to 0b to inform the user that the operation is completed.

2.4 AxiDMACtrl128

When the Read command is executed, AxiDMACtrl128 moves the data from AxiRxFifo to Host memory using a 128-bit AXI4 bus. Conversely, when the Write command is executed, it moves data from Host memory to AxiTxFifo. To facilitate data transfer between the Host system and each NVMeTCP25G-IP, the test application must allocate a buffer in Host Memory. This buffer is split into four areas (Buffer#0-#3) to enable quad buffering, allowing for parallel processing by the CPU and the hardware (AxiDMACtrl128). This setup enhances performance and increases more safe time gap for the CPU and the hardware to handle the data of each buffer area, as shown in Figure 2-12. However, if CPU and AxiDMACtrl128 can transfer data continuously without long pauses, using double buffer should be sufficient. The order to use the buffer is Buffer#0 -> #1 -> #2 -> #3 -> #0.

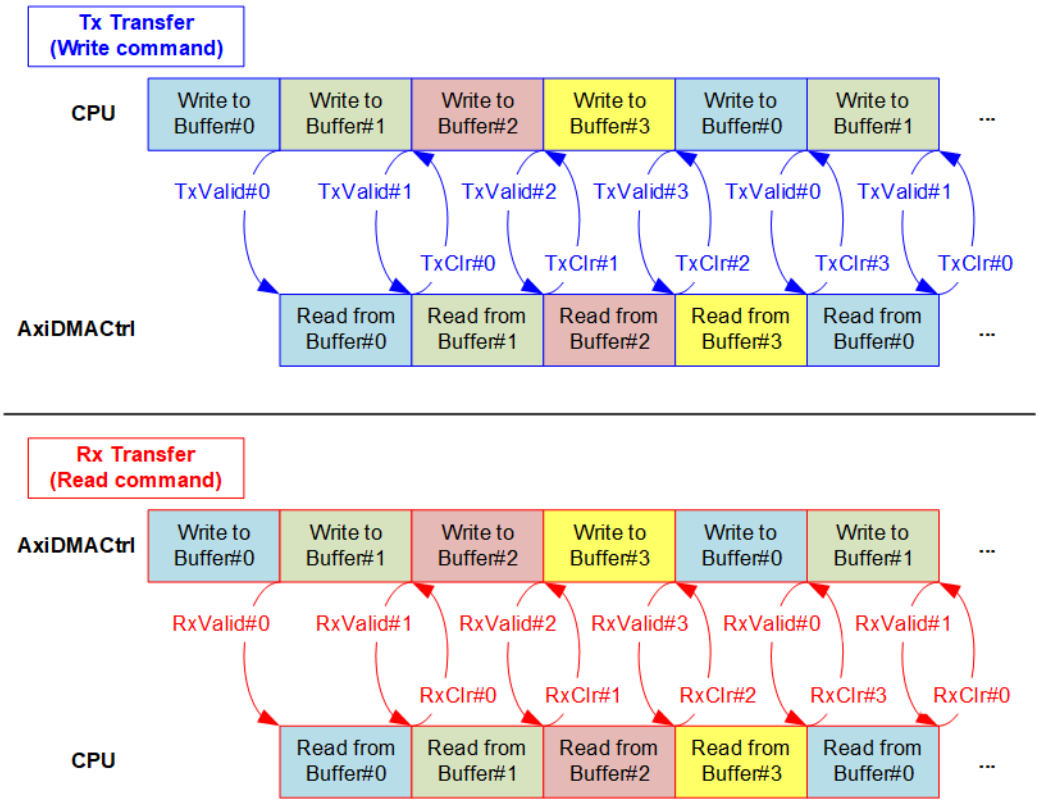

Figure 2-12 Quad buffering for DMA engine

For Tx transfer in the Write command, the Host memory is written by the test application on the host system and read by the DMA engine in the hardware system (AxiDMACtrl). The steps for this operation are outlined below.

1) To transfer data in the Write command, the CPU writes test data to the first area of the Host memory (Buffer#0). The Valid status of this area (TxValid#0) is asserted when this memory area becomes full or when the last data is written.
2) If there is remaining data to write, the CPU checks the status of the next area (Buffer#1). If it is free, the CPU starts writing the test data to the next area. In parallel, AxiDMACtrl detects that data of the new buffer area is available, indicated by TxValid#0 asserted, which causes the hardware to start reading the data from Buffer#0 for processing.
3) Once the last data of this buffer area is read by AxiDMACtrl, the Clear flag of the buffer (TxClr#0) is asserted by AxiDMACtrl to free the buffer. If all data is not received, AxiDMACtrl checks the status of the next memory area (TxValid#1). If the new data is available, AxiDMACtrl continues the read operation for the next area.

   The CPU and AxiDMACtrl execute the write and read processes in parallel, but they operate on different buffer areas by repeating step 2) – 3) until all data is transferred.

Similarly, for Rx transfer in the Read command, the process is the same as the Tx transfer, but the roles are reversed. AxiDMACtrl writes to the Buffer, and the test application reads from it.

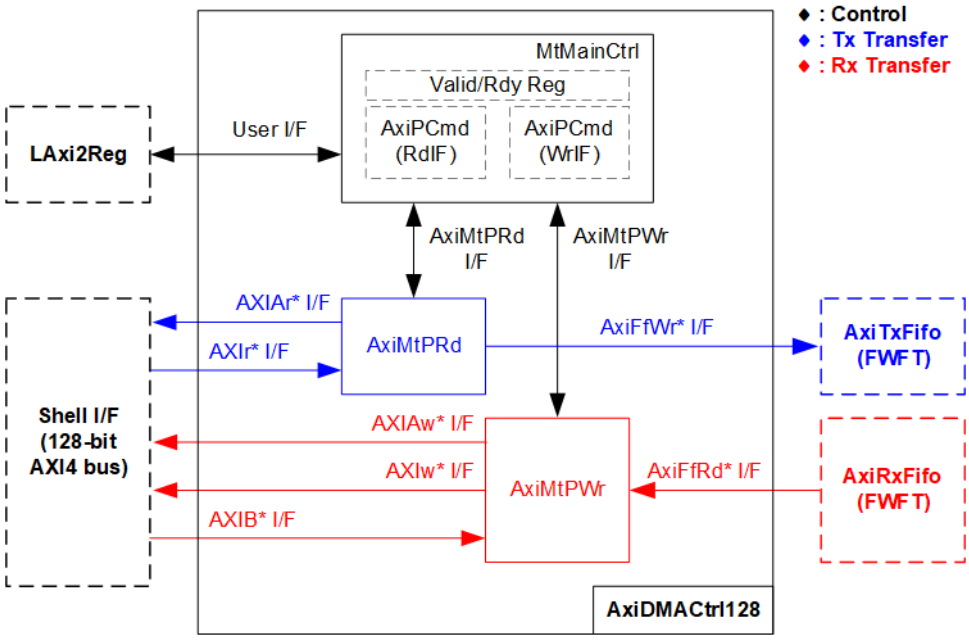

Figure 2-13 AxiDMACtrl512 Block Diagram

As shown in Figure 2-13, the AxiDMACtrl128 module has three submodules - MtMainCtrl, AxiMtPRd, and AxiMtPWr. These submodules are designed to facilitate separate operation for Tx and Rx transfers. The MtMainCtrl submodule contains registers for test parameters that are set by the CPU. Once the test parameters are decoded, MtMainCtrl generates a request with the relevant parameters for AxiMtPRd to initiate Tx transfer or for AxiMtPWr to initiate Rx transfer. AxiMtPRd is responsible for generating a memory read request to the host system via AXI4 I/F. It transfers data from the Host memory (Buffer#0-#3) to AxiTxFifo. On the other hand, AxiMtPWr generates a memory write request and writes data from AxiRxFifo to the Host memory (Buffer#0-#3) via AXI4 I/F. More detail of each submodule is described below.

2.4.1 MtMainCtrl

The function of the MtMainCtrl is to generate a command request to either AxiMtPWr or AxiMtPRd to initiate data transfer with each buffer area. If the total transfer size is too large to be handled by a single buffer area, multiple command requests are created for using multiple buffer areas. Both AxiMtPWr and AxiMtPRd have the same command interfaces, so the same modules, AxiPCmd, are applied to generate the command request for each individually. This section shows how AxiMtPCmd controls AxiMtPRd during Tx transfer and AxiMtPWr during Rx transfer using timing diagrams in Figure 2-14 and Figure 2-15, respectively.

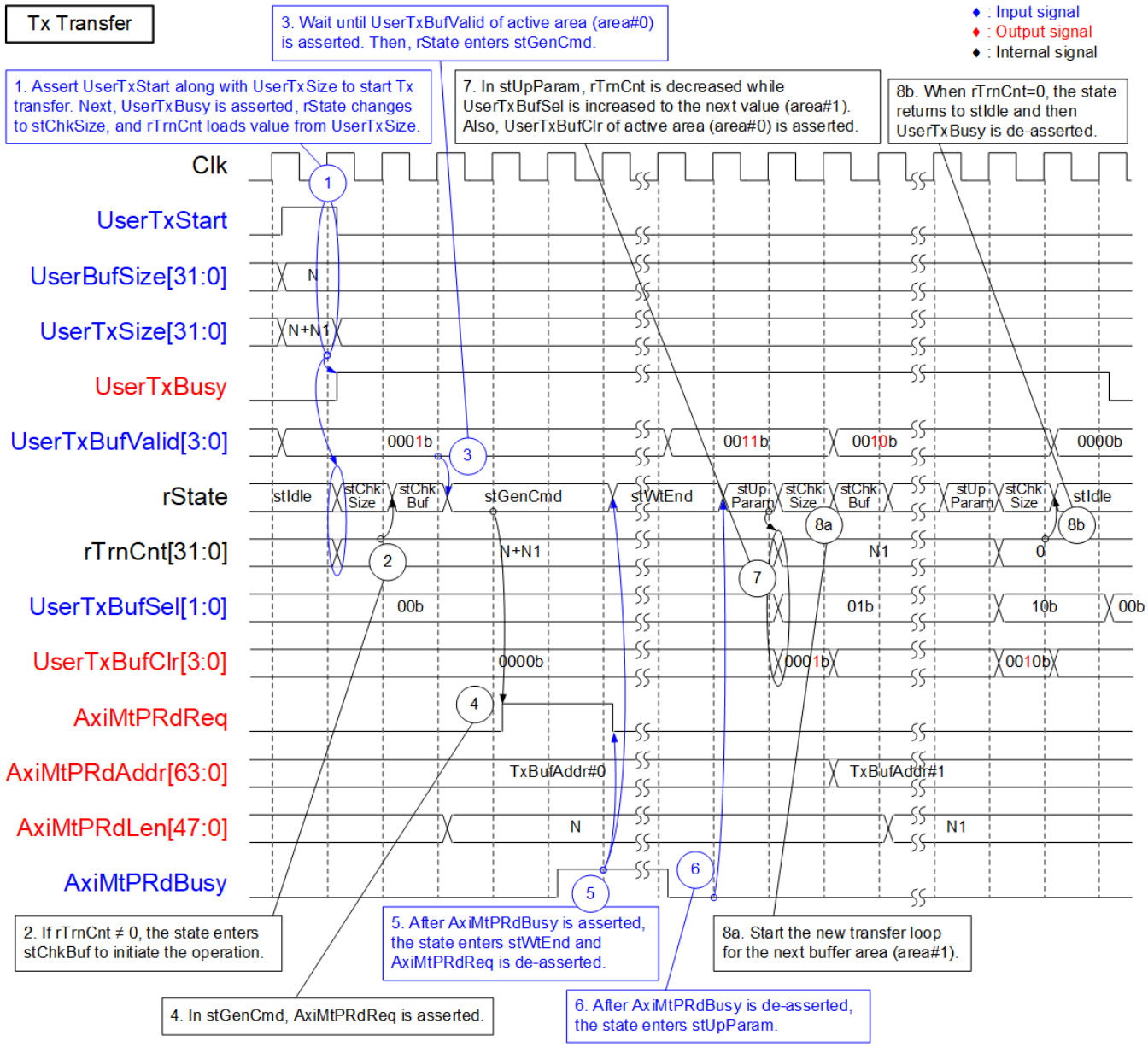

Figure 2-14 Tx transfer of MtMainCtrl timing diagram

In Figure 2-14, the CPU sends a Tx request to MtMainCtrl with a transfer size of N + N1, where N is the buffer size and N1 is less than or equal to N. AxiPCmd creates two command requests (AxiMtPRdReq) to AxiPRd, with request transfer size set to be N and N1, respectively. Each bit of UserTxBufValid signal is used to monitor the data flow of each buffer area for each command request, and each bit of UserTxBufClr signal is used to clear each buffer area. Four bits of valid and clear flags are used for the status of the four buffer areas (Buffer#0-#3).

The details of Figure 2-14 are as follows.

1) The CPU initiates the Tx transfer by asserting UserTxStart to MtMainCtrl with the buffer size (UserBufSize) and total transfer size (UserTxSize) parameters set. The first buffer area to operate is area#0 (Buffer#0), and the UserTxBufSel value indicating the active buffer area is reset to 00b. UserTxBusy is asserted to 1b to indicate that the Tx operation is processing, and the state enters stChkSize.
2) In stChkSize, the remaining transfer size (rTrnCnt) is read to determine the next state. If rTrnCnt is not equal to 0, the state continues to stChkBuf. Otherwise, the state returns to stIdle (step 8).
3) In ChkBuf, the buffer status (UserTxBufValid) of the active area is read. Bit[0] is read if the active area is area#0. The state waits until UserTxBufValid is set to 1b before entering stGenCmd.
4) In stGenCmd, AxiMtPRd I/F parameters (AxiMtPRdAddr and AxiMtPRdLen) are generated, with AxiMtPRdAddr being the start buffer address and AxiMtPRdLen being the transfer size. AxiMtPRdReq is asserted to AxiMtPRd. AxiMtPRdLen is always equal to UserBufSize (N) for each run loop, except the last loop where it is equal to the remaining value (N1).
5) While the command is processing, AxiPRd asserts AxiPRdBusy to 1b, and the state enters stWtEnd to wait until AxiPRd completes the process.
6) Once AxiPRd finishes processing, AxiPRdBusy is set to 0b, and the state enters the last state, stUpParam.
7) In stUpParam, the internal signals update their values for the next transfer. rTrnCnt decreases to indicate the remaining transfer length, UserTxBufSel is increased to indicate the next active buffer area, and UserTxBufClr of the active area (area#0) is set to 1b for one cycle to clear the UserTxBufValid flag. As a result, UserTxBufValid of the active area is cleared in the next clock cycle. The state then returns to stChkSize to check if the operation is completed or not.
8) The next state is determined by the value of rTrnCnt as described in step 2).
   a) If rTrnCnt ≠ 0, step 2) – 7) are repeated to transfer data using the next buffer area (in order #0 -> #1 -> #2 -> #3 -> #0).
   b) If rTrnCnt = 0, the operation is completed, and the state returns to stIdle. UserTxBusy is then set to 0b.

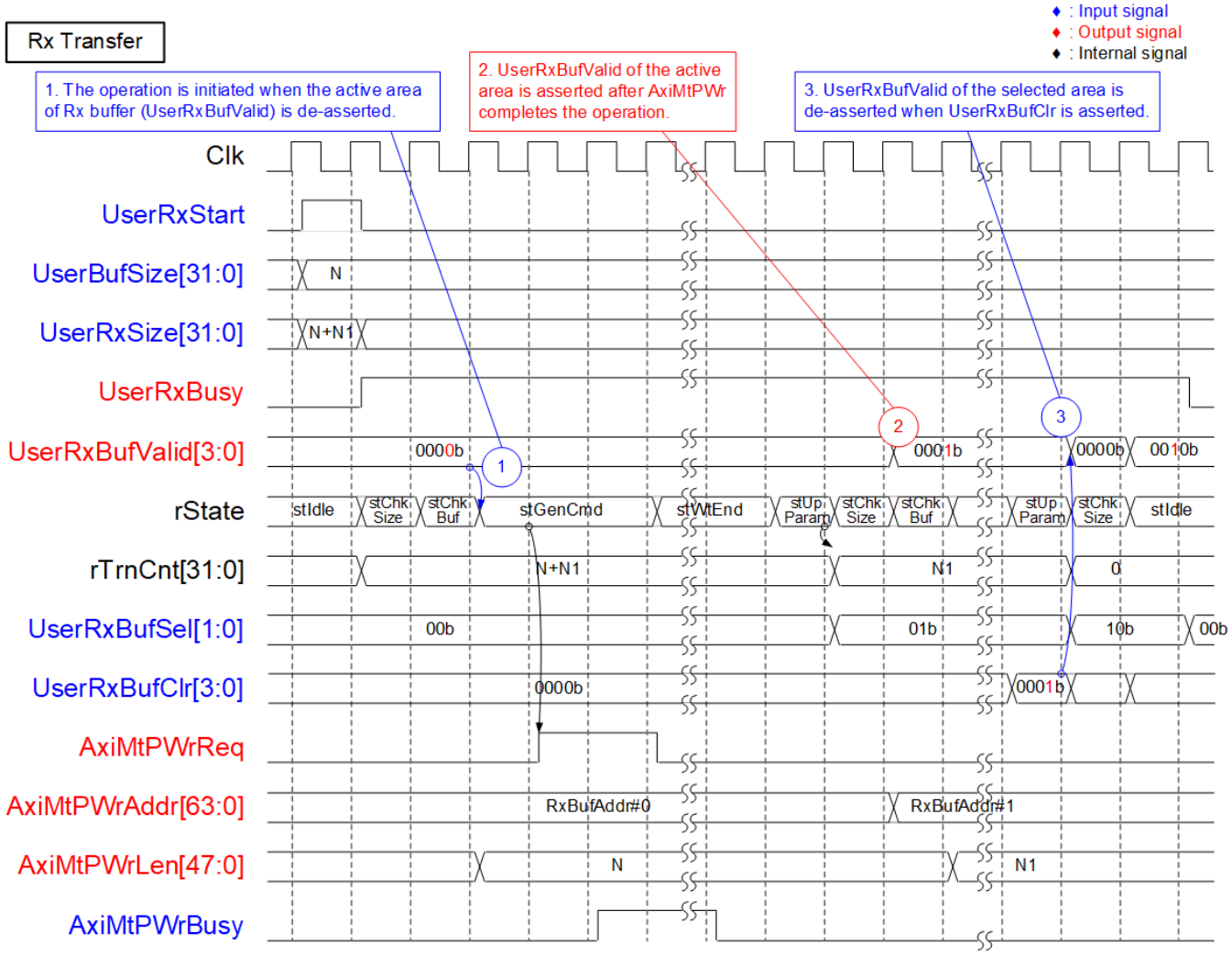

Figure 2-15 Rx transfer of MtMainCtrl timing diagram

Figure 2-15 demonstrates the Rx operation which has the same request flow as the Tx operation shown in Figure 2-14, except for the direction of data transfer which is reversed. During Rx transfer, UserRxBufValid is set after writing data to each buffer area, while UserRxBufClr is set by the CPU to free each buffer area after reading the data. Four bits of valid and clear flags are used to indicate the status of the four buffer areas (Buffer#0-#3). The details of the data flow during Rx transfer are as follows.

1) In stChkBuf, the buffer status (UserRxBufValid) of the active area is read. The state waits until UserRxBufValid is set to 0b to indicate that this area is free before entering stGenCmd. After that, AxiPMtWr writes the data the Host memory.
2) After AxiMtPWr finishes writing the data in each loop, indicated by AxiMtPWrBusy set from 1b to 0b, UserRxBufValid of the active area is set to 1b to indicate that the data is available for reading.
3) When the CPU finishes reading the data in that area, it sets UserRxBufClr to 1b to clear the UserRxBufValid flag. As a result, UserRxBufValid is cleared in the next clock cycle to indicate the free status.

2.4.2 AxiMtPRd

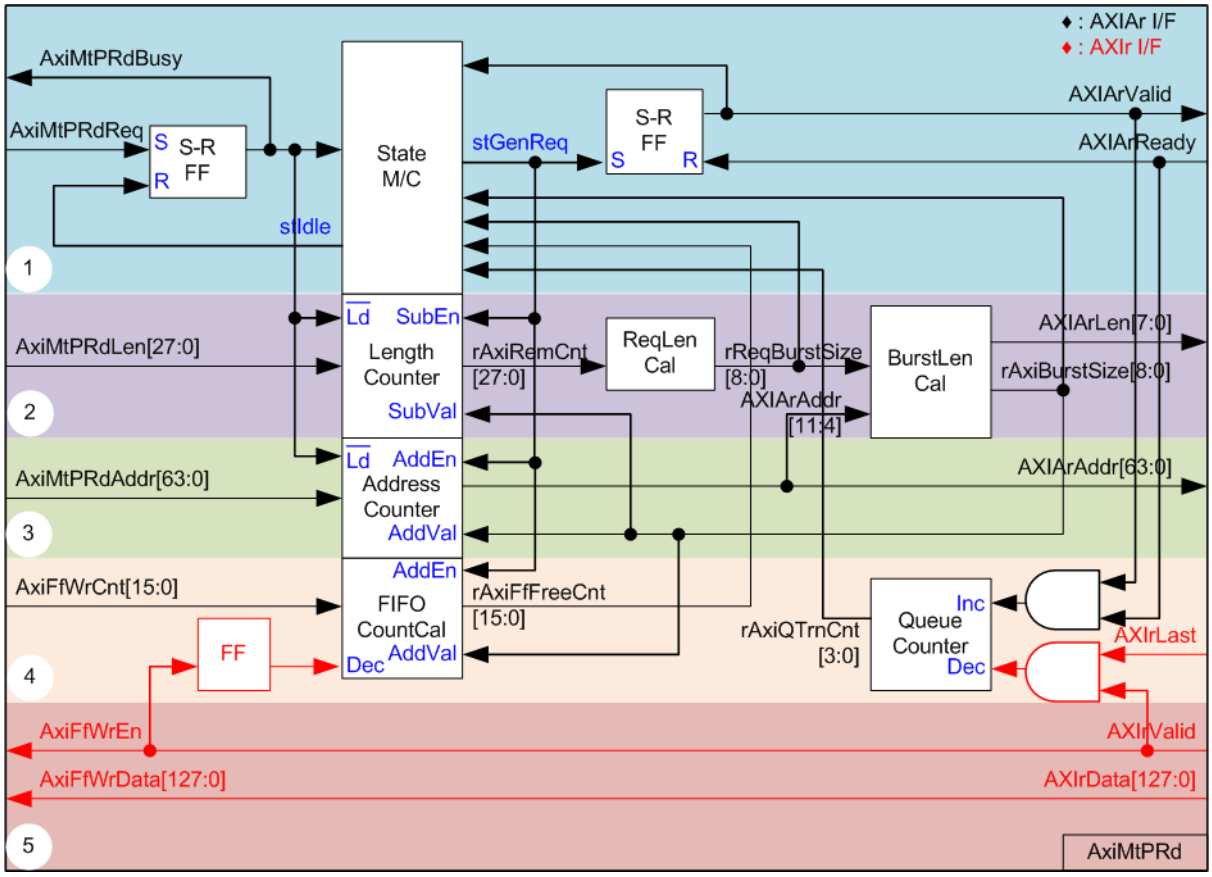

Figure 2-16 AxiMtPRd Block diagram

The operation of AxiMtPRd is initiated when MtMainCtrl asserts AxiMtPRdReq. AxiMtPRd generates a read command request to the Shell via AXIAr I/F and then the read data is returned from the Host memory via AXIr I/F. AxiMtPRd bypasses the received data from AXIr I/F to store in AxiRxFifo. According to AXI4 standard, AXIAr I/F and AXIr I/F can operated in parallel, so the logics for sending the new read command request and monitoring the read data are executed individually to perform the best performance.

To initiate the read operation, the user must assert AxiMtPRdReq along with AxiMtPRdLen (Total transfer length in 64-byte unit) and AxiMtPRdAddr (Start address of the Host memory in byte unit). The state machine in Block no.1 generates the read command request (AXIArValid). Block no.2 generates the transfer size (AXIArLen) of each command request, which can be 1, 32, or 256. The Length Counter (rAxiRemCnt) loads the total length size (AxiMtPRdLen) during initialization and calculates the remaining transfer size after generating each request to AXIAr I/F. The BurstLenCal block determines the actual transfer size (AXIArLen) based on the maximum transfer size of each request (rReqBurstSize) based on rAxiRemCnt and the current address (AXIArAddr[11:4]) to avoid the address boundary crossing problem in each data transfer. AXIArLen can be less than rReqBurstSize value if the current address is not aligned with the requested transfer size. Block no.3 is the address counter that calculates the next start address (AXIArAddr) after generating each command request to AXIAr I/F.

Block no.4 contains the flow control logic responsible for verifying two parameters before the state machine can send a new request. These parameters include the available free space size in AxiTxFifo (rAxiFfFreeCnt) and the number of advance command requests that have not yet received data (rAxiQTrnCnt). If there is enough free space in AxiTxFifo and the number of advance command requests has not yet reached the limit, the state machine can generate a new request. To compute the used FIFO size, FIFOCountCal adds the current value of the FIFO data counter (AxiFfWrCnt) to the amount of data that has been requested but not yet received, and then uses NOT logic to determine the free space size. Meanwhile, Queue Counter (rAxiQTrnCnt) is incremented when a new command request is asserted and decremented when the last data of each request is received. A four-bit Queue counter is applied to limit the maximum number of advance command requests to 15. This restriction is in place to prevent generating too many requests to the AXI4 interface. Finally, Block no.5 shows the data interface that bypasses data from Axir I/F to AxiFIFO I/F.

Figure 2-17 shows the timing diagram of AxiMtPRd logic when the new command request is set along with setting total transfer size to 257 and the start address on AxiPRdAddr to A0. Assumed that A0 is aligned to 64-byte unit. In this condition, AxiMtPRd generates two read command requests with setting burst size to 256 and 1, respectively, to AXIAr I/F, and then all data is returned to AxiMtPRd through AXIr I/F.

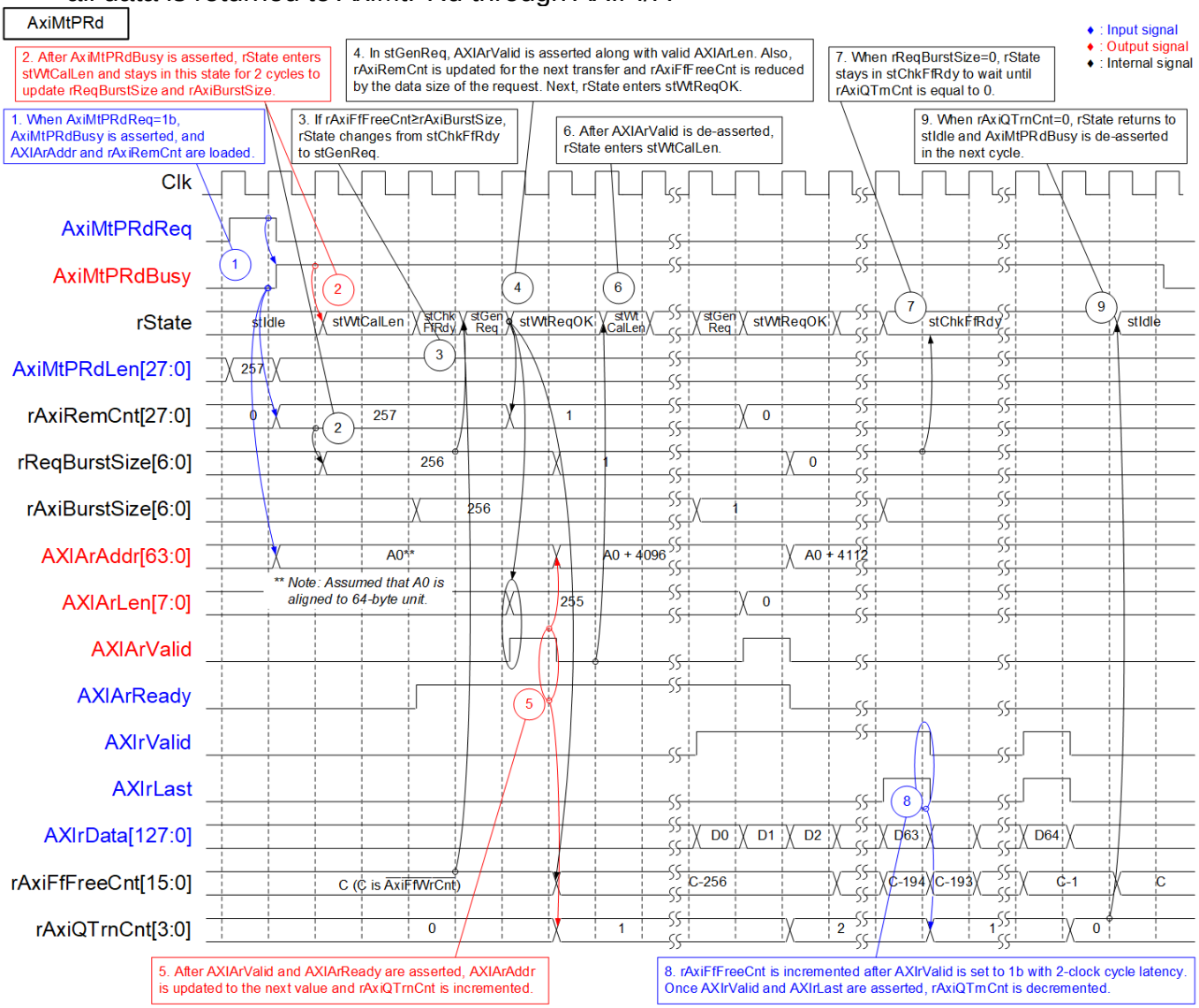

Figure 2-17 AxiMtPRd Timing diagram

1) To initiate the operation, the new command request (AxiMtPRdReq) must be set to 1b, along with the valid AxiMtPRdAddr (the start address of Host memory) and AxiMtPRdLen (total transfer size in 16-byte units). Upon acceptance of the request, AxiMtPRdBusy is set to 1b, and the initial values for AXIArAddr and AxiMtPRdAddr are loaded from AxiMtPRdAddr and AxiMtPRdLen, respectively.
2) When AxiMtPRdBusy is asserted, rState enters stWtCalLen to determine the transfer size of the command request to AXIAr I/F. If rAxiRemCnt is equal to or greater than 256, rReqBurstSize (which indicates the maximum request size) is set to 256; otherwise, it is set to rAxiRemCnt. Next, rAxiBurstSize is calculated using rReqBurstSize and the lower bits of AxiArAddr to determine the maximum transfer size which does not cross the address boundary. The state remains in stWtCalLen for two clock cycles to wait until rAxiBurstSize is updated before entering stChkFfRdy.
3) In stChkFfRdy, if there is remaining command request to generate, indicated by rReqBurstSize≠0, it waits until the FIFO has enough free space to store data for the next transfer (rAxiFfFreeCnt ≥ rAxiBurstSize) before entering stGenReq. Step 7 shows the condition that all command request has been generated.
4) In stGenReq, AXIArValid and valid AXIArLen (set to rAxiBurstSize – 1) are set, and rAxiRemCnt and rAxiFfFreeCnt are decreased by the transfer size of this request. The state then enters stWtReqOK.
5) In stWtReqOK, the state waits until AXIArReady is set to 1b, indicating that the command has been accepted by AXIAr I/F. Once AXIArReady is set to 1b, AXIArValid is de-asserted, AXIArAddr is increased by (AXIArLen x 16), and rAxiQTrnCnt is incremented.
6) After AXIArValid is de-asserted to 0b, the state returns to stWtCalLen to perform the operation described in step 2). Step 2) – 6) are repeated to generate the next command request until rReqBurstSize=0 in stChkFfRdy.
7) In stChkFfRdy, if rReqBurstSize=0, the state waits until all data is received from AXIr I/F, (indicated by rAXIQTrnCnt=0) before entering stIdle.
8) Once the command request is accepted by AXIAr I/F, data can be returned through AXIr I/F at any time. AXIrValid is set to 1b along with valid data on AXIrData for each data transmission. As each data is received, rAxiFfFreeCnt is incremented with a 2-clock cycle latency. When the last data of each command request is received (indicated by AXIrLast=1b and AXIrValid=1b), rAxiQTrnCnt is decremented.
9) Once all data is received, the state returns to stIdle and AxiRdBusy is set to 0b to allow the new command request set by user.

2.4.3 AxiMtPWr

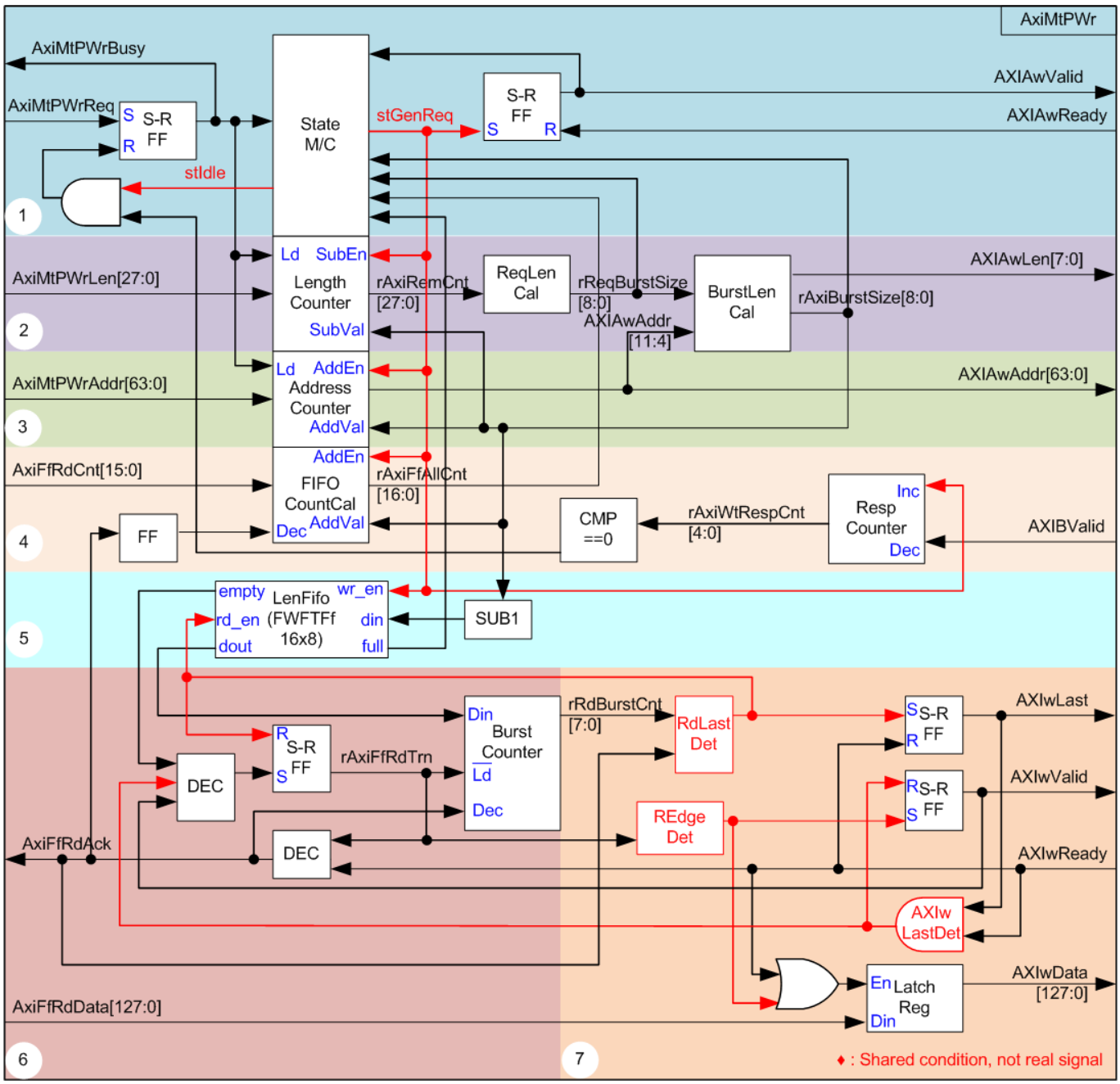

Figure 2-18 AxiMtPWr Block diagram

The data transfer direction of AxiMtPWr is reversed from AxiMtPRd. When the user sends a request, AxiMtPWr sends the write request to AXIAw I/F and the write data to AXIw I/F. AXIAw I/F and AXIw I/F can be controlled individually, and their logics are separately designed. As shown in Figure 2-18, AxiMtPWr is comprised of three logic groups: AXIAw I/F logics (Block no.1 – no.4), connecting logics between AXIAw I/F and AXIw I/F(Block no.5), and AXIw I/F logic (Block no.6 – no.7).

Block no.1 contains the state machine, which generates the write command request (AXIAwValid) to AXIAw I/F until all command requests have been generated. Additionally, this block creates the busy flag (AxiMtPWrBusy), which indicates that the command is processing. AxiMtPWrBusy is asserted after a new request is asserted (AxiMtPWrReq=1b), and it is de-asserted when the state returns to stIdle, meaning all command requests have been generated, and all command responses are returned through AXIB I/F (detailed in Block no.4). Block no.2 and Block no.3 generate the transfer size (AXIAwLen) and the start address (AXIAwAddr) of each command request, respectively. Both blocks use the same logic as Block no.2 and no.3 of AxiMtPWr, as described in section 2.4.2.

Block no.4 contains the logic to calculate two parameters. Firstly, it calculates the amount of data in the AxiRxFifo that has not been requested (rAxiFfAllCnt) from the number of data stored in AxiRxFifo (AxiFfRdCnt) being decreased by the amount of data that is requested but has not been transferred. The state must ensure that rAxiFfAllCnt≥rAxiBurstSize before generating the new request to AXIAw I/F. Secondly, it calculates the remaining number of command response that has not been received, rAxiWtRespCnt, which is the output of Resp Counter. rAxiWtRespCnt is incremented after the new request is generated to AXIAw I/F and decremented after each write response is received (AXIBValid=1b). If rAxiWtRespCnt = 0, it indicates that all command responses have been received, and the operation has been completed. The size of rAxiWtRespCnt is related to the LenFifo depth size, which limits the maximum request that can be generated without waiting for the command response to be returned.

Block no.5 contains LenFIFO which stores the transfer length of each command request sent to AXIAw I/F. The LenFIFO is First-Word Fall-Through (FWFT) type and read by the data interface logic inside Block no.6 to control the number of transmitted data to AXIw I/F to match the transfer size set in each command request. Block no.6 controls data transfer from AxiRxFifo to AXIw I/F. The core signal of this block is rAxiFfRdTrn, which is set to 1b while transmitting data. A new data transfer is initiated by setting rAxiFfRdTrn to 1b when LenFifo has data (empty of LenFifo=0b) and the data interface is Idle (AXIwValid=0b) or the last data has been sent (AXIwLast=1b and AXIwReady=1b). rAxiFfRdTrn is set to 0b when the last data is read from AxiRxFifo, monitored by RdLastDet. To track the amount of data in each transfer, Burst Counter is designed to count the remaining data size of each request. It loads the initial value from LenFIFO and then decreases the value after reading each data from AxiRxFifo (AxiFfRdAck=1b). The last data is transferred (RdLastDet is asserted) when rRdBurstCnt is equal to 0. If AXIw I/F is not ready to receive data, it de-asserts AXIwReady to 0b, which causes AxiFfRdAck to be de-asserted to pause the reading of the next data from AxiRxFifo.

Finally, Block no.7 contains the output registers of AXIw I/F signals. AXIwValid remains set to 1b while transmitting all data of each transfer because all data of that transfer is available in AxiRxFifo before sending the request to AxiAw I/F. AXIwValid is set to 1b after the new transfer is initiated by setting rAxiFfRdTrn from 0b to 1b, and set to 0b after the final data is completely sent (AXIwLast=1b and AXIwReady=1b). The data interface signals of AxiRxFifo are fed to a Flip-Flop or Register before forwarding them to AXIw I/F.

Figure 2-19 and Figure 2-20 show timing diagram of the command interface (AXIAw I/F) and the data interface (AXIw I/F) when AxiMtPWr is executed. User sets transfer length to 257 for this request, so two write command requests are created to AXIAw I/F with setting the burst size to 256 and 1, respectively. Assume that the start address of the user request is aligned to 64-byte unit. The core engine for controlling AXIAw I/F is the state machine.

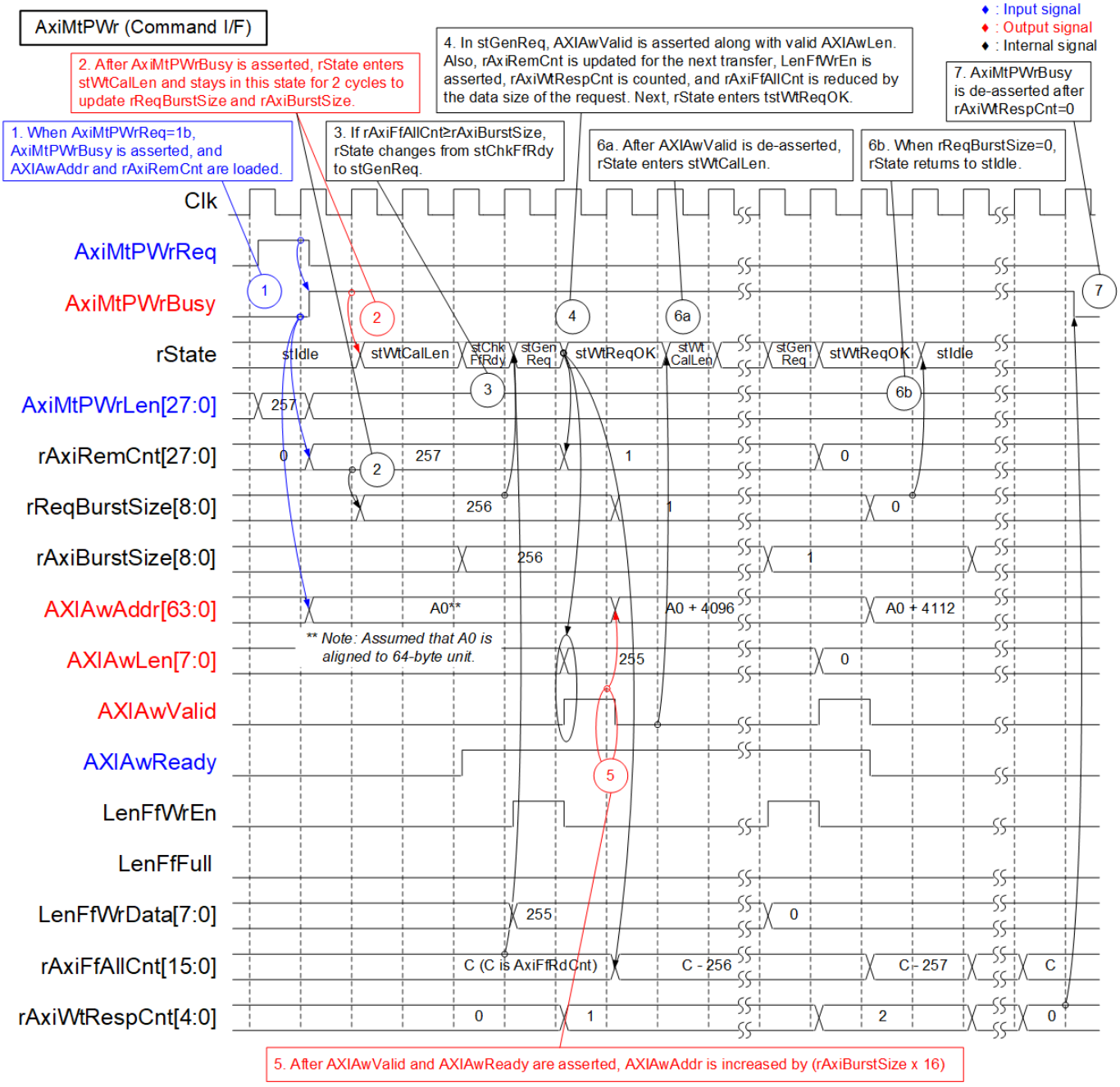

Figure 2-19 Command I/F of AxiMtPWr Timing diagram

Steps 1) – 5) for generating the write request to the AXIAw I/F in Figure 2-19 are similar to those described in steps 1) – 5) of Figure 2-17 in section 2.4.2. However, the AXIAr I/F signals and rAxiFfFreeCnt are replaced by the AXIAw I/F signals and rAxiFfAllCnt, respectively. rAxiWtRespCnt, which has the same funcation as rAxiQTrnCnt, is updated in step 4) instead of step 5).

The LenFifo stores the burst size of each request when the state is stGenReq. Therefore, in stChkFfRdy, the full flag of LenFifo (LenFfFull) is checked to ensure that the FIFO is not full before the state enters stGenReq. LenFfWrData is equal to rAxiBurstSize – 1.
Note: Figure 2-19 does not include the data interface, so the operation of rAxiFfAllCnt is not completely displayed. When each data is read from AxiRxFifo (AxiFfRdAck=1b), rAxiFfAllCnt is incremented. After finishing the operation, the last value of rAxiFfAllCnt is equal to AxiFfRdCnt.

6) The next state after stWtReqOK is determined by the value of rReqBurstSize, which is read when AXIAwValid is de-asserted to 0b to ensure that the command request has been accepted.
   a) If rReqBurstSize≠0, which means there are remaining command requests for generating, it returns to stWtCalLen, and steps 1) – 6) are repeated to operate the next request.
   b) If rReqBurstSize=0, rState returns to stIdle.
7) To ensure that all command responses are received, rAxiWtRespCnt is applied. If it is equal to 0, the operation is completed, and the state returns to stIdle. AxiWrBusy is then set to 0b to allow the new command request set by user.

When LenFifo has data, the data interface on AXIw I/F is initiated by reading the transfer size of each packet from LenFifo. After that, the data from AxiRxFifo which is FWFT type is forwarded to AXIw I/F. Two transfer sizes are demonstrated in Figure 2-20, 256 and 1, respectively.

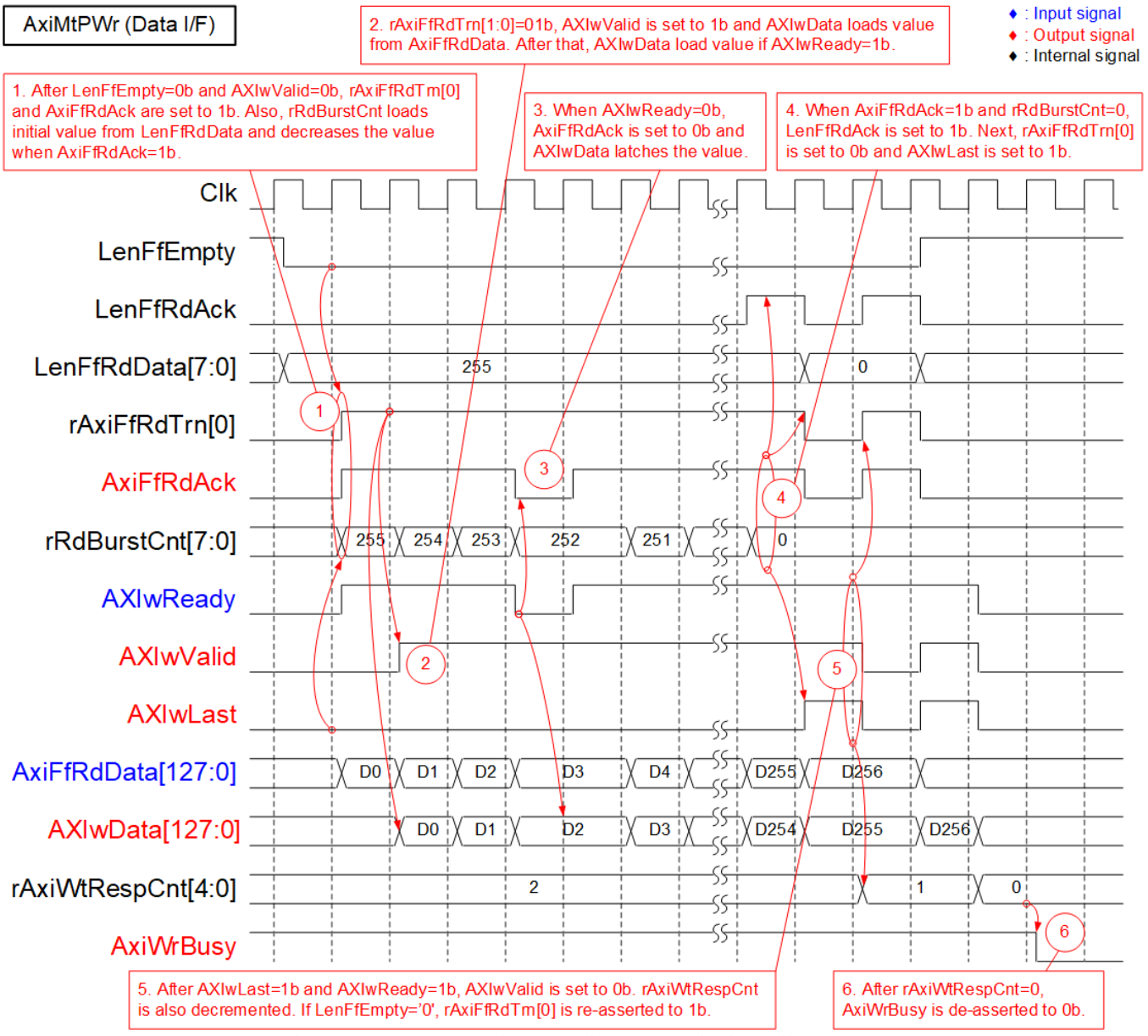

Figure 2-20 Data I/F of AxiMtPWr Timing diagram

1) The data transfer is initiated when LenFifo has data (LenFfEmpty=0b) and there is no ongoing data transfer on the AXIw I/F (AXIwValid=0b). The transfer starts by setting rAxiFfRdTrn[0] to 1b, which remains asserted until the current data transfer is completed. The data is read from AxiRxFifo by setting AxiFfRdAck to 1b. The amount of transferred data is tracked using the burst counter (rRdBurstCnt), which is initialized with the value from LenFifo's data output (LenFfRdData) and decremented when AxiFfRdAck is set to 1b.
   Note: The next data transfer can be initiated without waiting for AXIwValid to be de-asserted to 0b, and it can also be initiated after the last data is transferred, as described in step 5).
2) After setting rAxiFfRdTrn to 1b to transfer the first data, AXIwValid is set to 1b and remains set until the last data is transferred to AXIw I/F. AXIwData loads the first data from AxiFfRdData to send the first data, and then it loads the new value when AXIwReady is set to 1b.
3) When AXIw I/F is not ready to receive new data, it de-asserts AXIwReady to 0b, causing the data transmission to pause by setting AxiFfRdAck to 0b. Additionally, AXIwData and other AXIw I/F signals hold the same value until AXIwReady is re-asserted to 1b.
4) When the last data is read from AxiRxFifo (AxiFfRdAck=1b and rRdBurstCnt=0), LenFfRdAck is asserted to 1b to flush the current data and read the next data from LenFifo. In the next clock, rAxiFfRdTrn[0] is set to 0b, and the last data of this transfer is forwarded to AXIw I/F (AXIwLast=1b and AXIwData=D255).
5) After the last data is accepted by AXIw I/F (AXIwLast=1b and AXIwReady=1b), AXIwValid is de-asserted to 0b, and rAxiWtRespCnt is decremented. At the same time, the new data transfer is initiated by repeating steps 1) – 5) if LenFifo has data (LenFfEmpty=0b). In Figure 2-20, the next data transfer sets the transfer size to be one.
6) After all data is completely transferred, as indicated by rAxiWtRespCnt being equal to zero, the busy flag (AxiWrBusy) is de-asserted to 0b, and the user can request a new command.

2.5 LAxi2Reg

AXI4-Lite is the interface of the hardware kernel for accessing hardware registers. This interface is used by the CPU to configure the hardware parameters and monitor the hardware's status during operation. It has a 32-bit data bus. LAxi2Reg operates on the application clock domain which its frequency is configured by the tool.

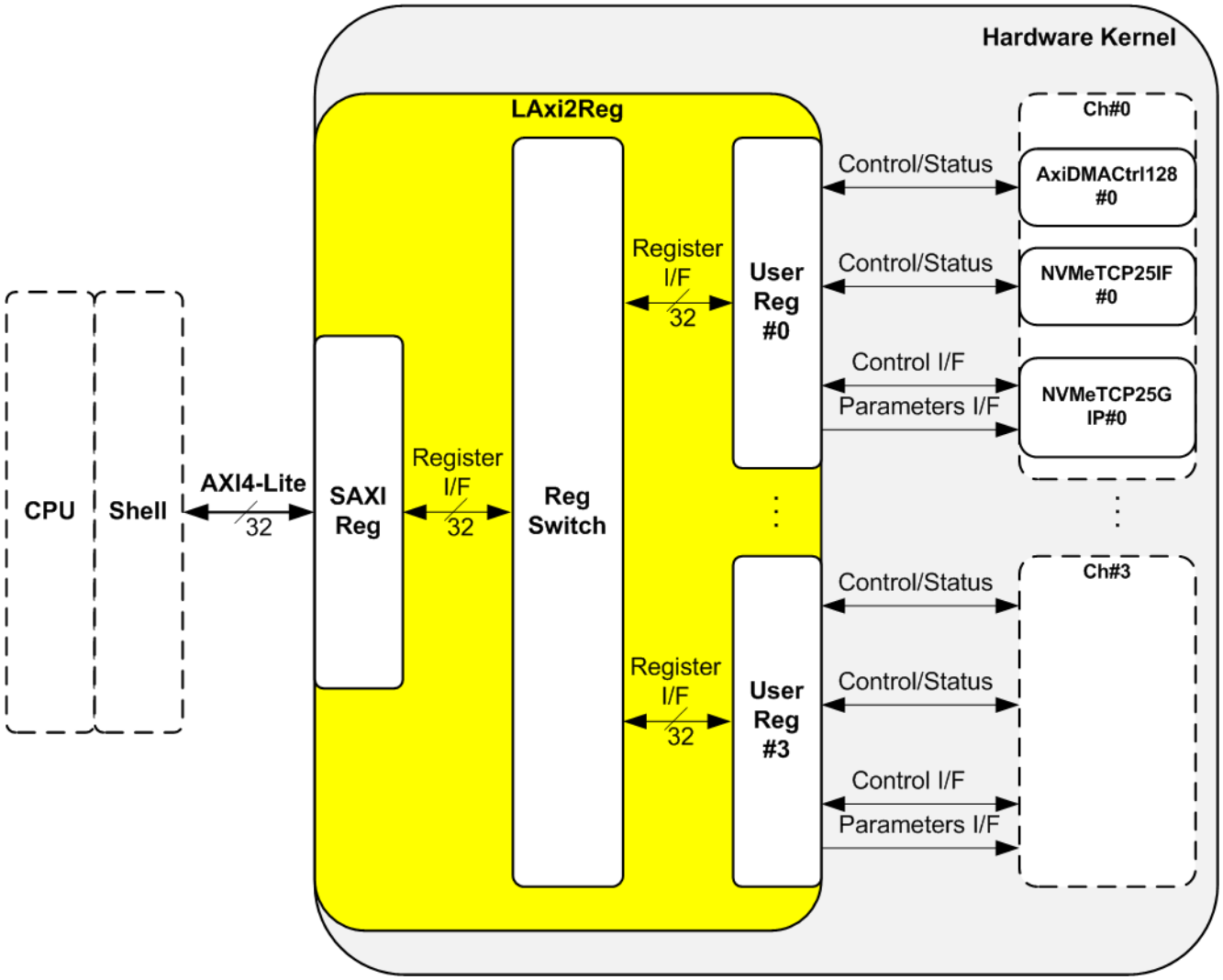

Figure 2-21 LAxi2Reg interface

LAxi2Reg is comprised of three components: SAXIReg, RegSwitch, and UserReg. SAXIReg converts the AXI4-Lite signals to a simple register interface with a 32-bit data bus size, which is similar to the AXI4-Lite data bus size. RegSwitch determines the active UserReg by decoding the upper bits of the address that is requested from SAXIReg. UserReg contains the register file for the parameters and the status of the submodules. More details on SAXIReg, RegSwitch, and UserReg are provided below.

2.5.1 SAXIReg

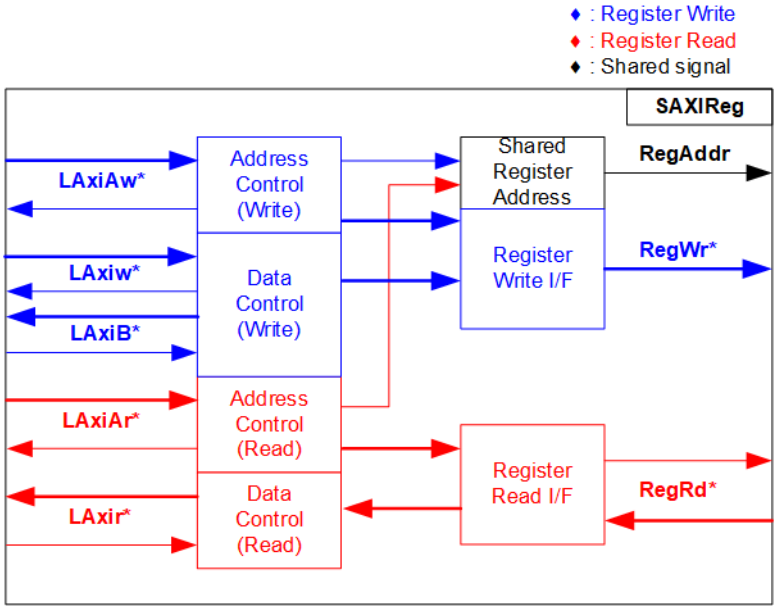

Figure 2-22 SAXIReg Interface

The AXI4-Lite bus interface signal can be divided 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). Additional details on building custom logic for the AXI4-Lite bus can be found in the following document. https://github.com/Architech-Silica/Designing-a-Custom-AXI-Slave-Peripheral/blob/master/designing_a_custom_axi_slave_rev1.pdf

According to the AXI4-Lite standard, the write and read channels operate independently, and the control and data interfaces for each channel are run separately. The logic inside SAXIReg for interfacing with the AXI4-Lite bus is divided into four groups: Write control logic, Write data logic, Read control logic, and Read data logic. The left side of Figure 2-22 illustrates this division. The Write control I/F and Write data I/F of the AXI4-Lite bus are latched and transferred to the Write register interface. Similarly, the Read control I/F of the AXI4-Lite bus is latched and transferred to the Read register interface, while the returned data from the Register Read I/F is transferred to the AXI4-Lite bus. The RegAddr signal in the Register interface is shared for write and read access. It loads the address from LAxiAw for write access or LAxiAr for read access.

The simple register interface is compatible with the single-port RAM interface for write transaction. The read transaction of the Register interface is slightly modified from the RAM interface by adding RdReq and RdValid signals to control the read latency time. Since the address of the register interface is shared for write and read transactions, the user cannot write and read the register at the same time. The timing diagram of the Register interface is shown in Figure 2-23.

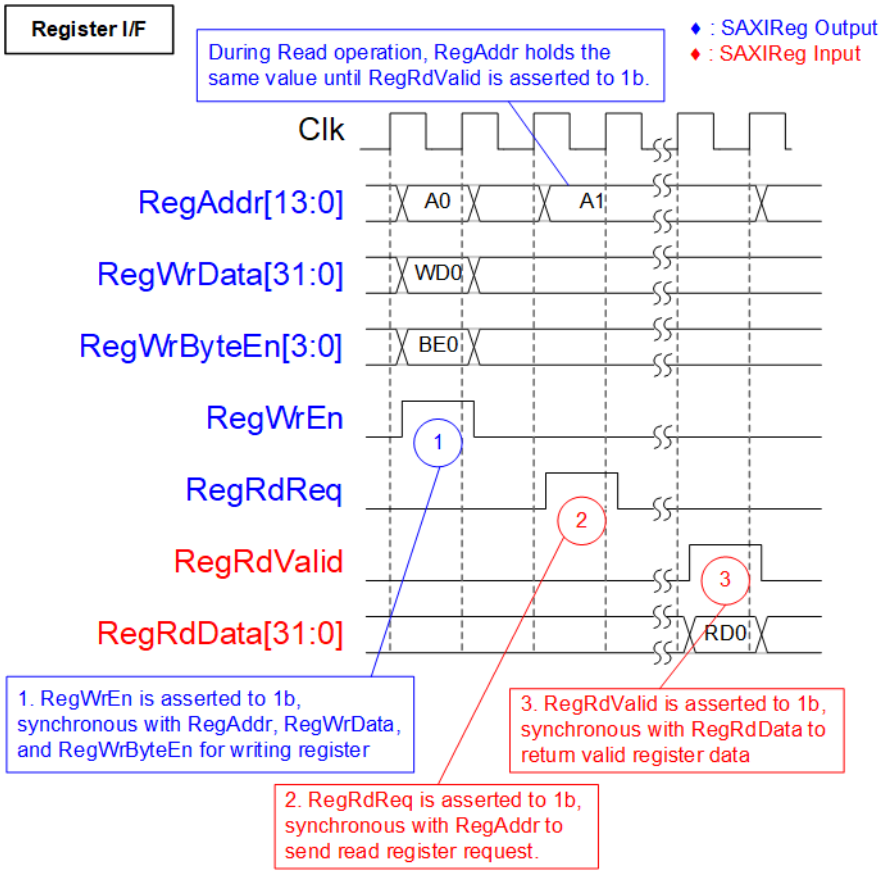

Figure 2-23 Register interface timing diagram

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

2.5.2 RegSwitch

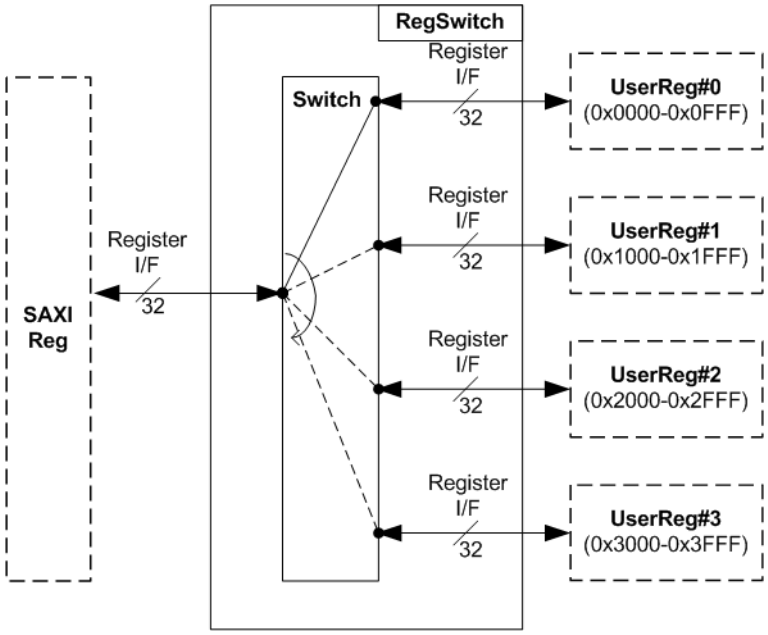

Figure 2-24 RegSwitch block diagram

RegSwitch is applied to mapped the Register I/F of SAXIReg to the Register I/F of the active UserReg. To determine the active UserReg, two bits of the address that is requested from SAXIReg are decoded because four UserREg modules are mapped to different address areas, as described below.

1) 0x0000 – 0x0FFF: UserReg#0
2) 0x1000 – 0x1FFF: UserReg#1
3) 0x2000 – 0x2FFF: UserReg#2
4) 0x3000 – 0x3FFF: UserReg#3

2.5.3 UserReg

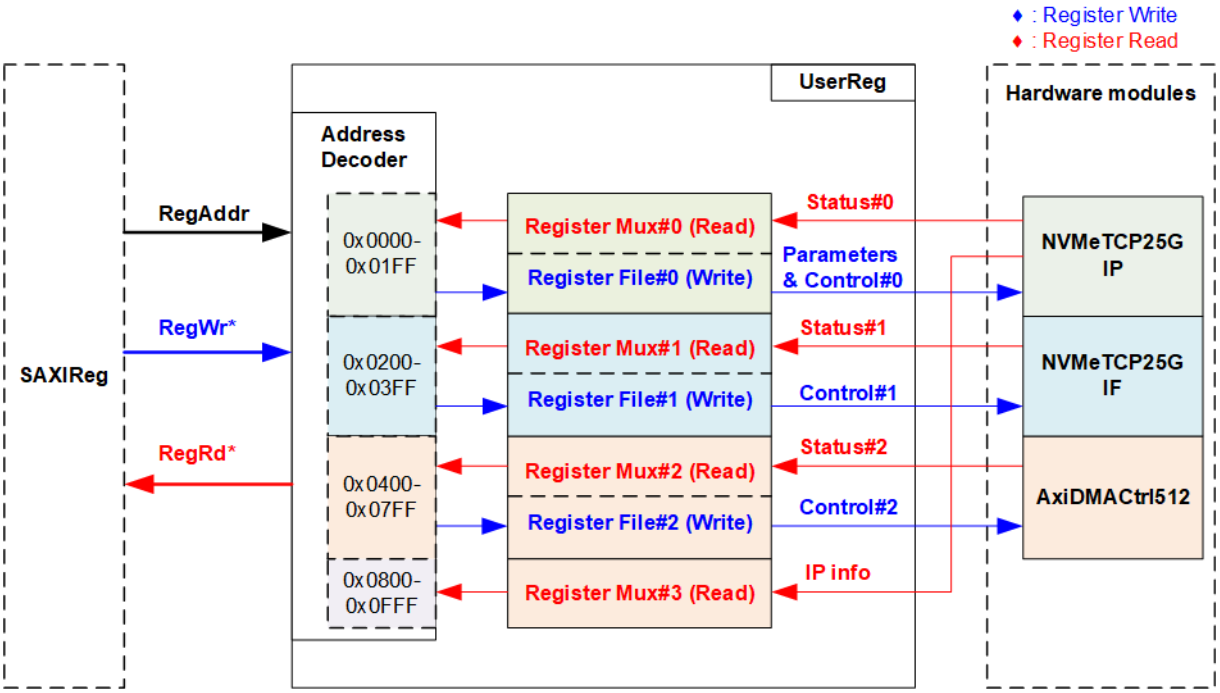

Figure 2-25 UserReg block diagram

UserReg contains registers that interface with various hardware submodules, including NVMeTCP25G-IP, NVMeTCP25IF, and AxiDMACtrl128. To select the active register, the address for write or read access is decoded by the Address decoder. Four addressing areas are available, as shown in Figure 2-25.

1) 0x0000 – 0x01FF: NVMeTCP25G-IP signal interfaces - Parameter, Control, and Status
2) 0x0200 – 0x03FF: NVMeTCP25IF control and status signals
3) 0x0400 – 0x07FF: AxiDMACtrl512 control and status signals
4) 0x0800 – 0x0FFF: IP information

To select the active address area, the upper bits of RegAddr are decoded by the Address decoder, while the lower bits are applied to select the active register in each area. The register file within UserReg is 32-bit data size, and write byte enable (RegWrByteEn) is not used, requiring the use of a 32-bit pointer by the CPU for writing these registers.

UserReg contains many status registers, and multi-level multiplexers are applied to return the read value. The latency time of read data is equal to four clock cycles, so RegRdValid is created by RegRdReq with asserting four D Flip-flops. Further information about the address mapping within UserReg module is presented in Table 2-1.

Table 2-1 Register map Definition

Address Wr/Rd	Register Name (Label in the NVMeTCP25DMATest.cpp) Description
BA+0x0000 – BA+0x01FF: NVMeTCP25G Parameters, Control, and Status interface (Write/Read access)	
BA+0x0000	Host MAC address (Low) DG_NVMETCP_TCP_HML_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be 32 lower bits of host MAC address (mapped to HostMAC[31:0] of NVMeTCP25G-IP)
BA+0x0004	Host MAC address (High) DG_NVMETCP_TCP_HMH_INTREG_OFFSET
	Wr/Rd - [15:0]: Input to be 16 upper bits of host MAC address (mapped to HostMAC[47:32] of NVMeTCP25G-IP)
BA+0x0008	Host IP address DG_NVMETCP_TCP_HIP_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be host IP address (mapped to HostIPAddr[31:0] of NVMeTCP25G-IP)
BA+0x000C	Host Port Number DG_NVMETCP_TCP_HPN_INTREG_OFFSET
	Wr/Rd - [15:0]: Input to be host admin port number (mapped to HostAdmPort[15:0] of NVMeTCP25G-IP) Wr/Rd - [31:16]: Input to be host I/O port number (mapped to HostIOPort[15:0] of NVMeTCP25G-IP)
BA+0x0010	Target MAC address (Low) DG_NVMETCP_TCP_TML_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be 32 lower bits of Target MAC address (mapped to TrgMAC[31:0] of NVMeTCP25G-IP)
BA+0x0014	Target MAC address (High) DG_NVMETCP_TCP_TMH_INTREG_OFFSET
	Wr/Rd - [15:0]: Input to be 16 upper bits of Target MAC address (mapped to TrgMAC[47:32] of NVMeTCP25G-IP)
BA+0x0018	Target IP address DG_NVMETCP_TCP_TIP_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be Target IP address (mapped to TrgIPAddr [31:0] of NVMeTCP25G-IP)
BA+0x0020	TCP Timeout DG_NVMETCP_TCP_TMO_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be TCP timeout value (mapped to TCPTimeOutSet[31:0] of NVMeTCP25G-IP)
BA+0x0024	NVMe Timeout DG_NVMETCP_NVME_TMO_INTREG_OFFSET
	Wr/Rd - [31:0]: Input to be NVMe timeout value (mapped to NVMeTimeOutSet[31:0] of NVMeTCP25G-IP)
BA+0x0040- Ba+0x004F	Host NQN Word 0-3 DG_NVMETCP_HSTNQN_INTREG_OFFSET(0-3)
	Input to be NVMe Qualifed Name (NQN) of the host (mapped to HostNQN [127:0] of NVMeTCP25G-IP) Wr/Rd - 0x0040: HostNQN [31:0], 0x0044: HostNQN[63:32],…, 0x004C: HostNQN[127:96]
BA+0x0080- Ba+0x008F	Target NQN Word 0-3 DG_NVMETCP_TRGNQN_INTREG_OFFSET(0-3)
	Input to be NVMe Qualifed Name (NQN) of the target (mapped to TrgNQN [127:0] of NVMeTCP25G-IP) Wr/Rd - 0x0080: TrgNQN [31:0], 0x0084: TrgNQN[63:32],…, 0x008C: TrgNQN[127:96]
BA+0x0100	Host connection status DG_NVMETCP_HCONNSTS_INTREG_OFFSET
	Wr – [0]: Input to enable the connection with the target (mapped to HostConnEn of NVMeTCP25G-IP) Rd – [0]: Ethernet linkup status from 25G Ethernet MAC (0b- Not linkup, 1b- Linkup). Rd – [1]: Mapped to HostConnStatus of NVMeTCP25G-IP (0b-Connection OFF, 1b-Connection ON) Rd – [2]: Mapped to HostBusy of NVMeTCP25G-IP (0b-IP is Idle, 1b-IP is busy) Rd - [3]: Mapped to HostError of NVMeTCP25G-IP (0b-No error, 1b-Error is found)
BA+0x0110	Total disk size (Low) DG_NVMETCP_LBASIZEL_INTREG_OFFSET
	Rd - [31:0]: Mapped to TrgLBASize[31:0] of NVMeTCP25G-IP
BA+0x0114	Total disk size (High) DG_NVMETCP_LBASIZEH_INTREG_OFFSET
	Rd - [15:0]: Mapped to TrgLBASize[47:32] of NVMeTCP25G-IP

Address Wr/Rd	Register Name (Label in the NVMeTCP25DMATest.cpp) Description
BA+0x0000 – BA+0x01FF: NVMeTCP25G Parameters, Control, and Status interface (Write/Read access)	
BA+0x0120	Capability (Low) Status DG_NVMETCP_CAPSTSL_INTREG_OFFSET Rd - [31:0]: Mapped to TrgCAPStatus[31:0] of NVMeTCP25G-IP
BA+0x0124	Capability (High) Status DG_NVMETCP_CAPSTSH_INTREG_OFFSET Rd - [15:0]: Mapped to TrgCAPStatus[47:32] of NVMeTCP25G-IP
BA+0x0130	Host Error Type DG_NVMETCP_HERRTYPE_INTREG_OFFSET Rd – [31:0]: Mapped to HostErrorType[31:0] of NVMeTCP25G-IP
BA+0x0140	NVMe Completion Status DG_NVMETCP_NVMCOMPSTS_INTREG_OFFSET Rd - [15:0]: Mapped to TrgAdmStatus[15:0] of NVMeTCP25G-IP Rd - [31:16]: Mapped to TrgIOStatus[15:0] of NVMeTCP25G-IP
BA+0x0150 - BA+0x015F	Test pin Word0-3 DG_NVMETCP_NVMTESTPIN_INTREG_OFFSET(0-3) Mapped to TestPin[127:0] of NVMeTCP25G-IP. Rd - 0x0150: TestPin[31:0], 0x0154: TestPin[63:32], … , 0x015C: TestPin[127:96]
BA+0x0200 – BA+0x03FF: NVMeTCP25IF Control and Status (Write/Read access)	
BA+0x0200	NVMeTCP25IF control DG_NVMEIF_CONTROL_OFFSET Wr – [0]: Set 1b to start the write/read operation of NVMeTCP25IF module. This flag is auto-cleared Rd – [0]: Busy status from NVMeTCP25IF (0b-Idle, 1b-Processing).
BA+0x0204	NVMeTCP25IF command DG_NVMEIF_COMMAND_OFFSET Wr/Rd – [0]: Select command to NVMeTCP25IF (0b-Write command, 1b-Read command)
BA+0x0220	NVMeTCP25IF transfer length (Low) DG_NVMEIF_LOW_TOTAL_TRANSFER_LENGTH_OFFSET Wr/Rd – [31:0]: 32 lower bits of transfer size for the Write/Read command in 512-byte unit
BA+0x0224	NVMeTCP25IF transfer length (High) DG_NVMEIF_HIGH_TOTAL_TRANSFER_LENGTH_OFFSET Wr/Rd – [15:0]: 16 upper bits of transfer size for the Write/Read command in 512-byte unit
BA+0x0228	NVMeTCP25IF start address (Low) DG_NVMEIF_LOW_START_ADDRESS_OFFSET Wr/Rd – [31:0]: 32 lower bits of start address for the Write/Read command in 512-byte unit
BA+0x022C	NVMeTCP25IF start address (High) DG_NVMEIF_HIGH_START_ADDRESS_OFFSET Wr/Rd – [15:0]: 16 upper bits of start address for the Write/Read command in 512-byte unit
BA+0x0240	NVMeTCP25IF current transfer length (Low) DG_NVMEIF_LOW_CURRENT_TRANSFER_LENGTH_OFFSET Rd – [31:0]: 32 lower bits of current transfer size in byte unit while executing Write/Read command
BA+0x0244	NVMeTCP25IF current transfer length (High) DG_NVMEIF_HIGH_CURRENT_TRANSFER_LENGTH_OFFSET Rd – [15:0]: 16 upper bits of current transfer size in byte unit while executing Write/Read command

Address Wr/Rd	Register Name (Label in the NVMeTCP25DMATest.cpp) Description
BA+0x0400 – BA+0x07FF: AxiDMACtrl128 Control and Status (Write/Read access) ***Note: BA+0x0600 – BA+0x06FF: Tx buffer parameters [CPU -> Hardware]*** ***BA+0x0700 – BA+0x07FF: Rx buffer parameters [Hardware -> CPU]***	
BA+0x0400	AxiDMACtrl128 reset DG_DMA_RESET_OFFSET
	Wr/Rd – [0]: Reset signal to AxiDMACtrl128 module (1b-Reset, 0b-Clear).
BA+0x0404	AxiDMACtrl512 command DG_DMA_COMMAND_OFFSET
	Wr – [0]: Start Tx transfer. Set to 1b to start Tx transfer on AxiDMACtrl128. This flag is auto-cleared. Wr – [1]: Start Rx transfer. Set to 1b to start Rx transfer on AxiDMACtrl128. This flag is auto-cleared.
BA+0x0408	AxiDMACtrl128 status DG_DMA_STATUS_OFFSET
	Rd – [0]: Tx transfer busy flag. Set to 1b while AxiDMACtrl128 is operating Tx transfer. Rd – [1]: Rx transfer busy flag. Set to 1b while AxiDMACtrl128 is operating Rx transfer. Rd – [9:8]: The active area of Tx buffer that is operating. (00b-Tx buffer#0, 01b-Tx buffer#1, 10b-Tx buffer#2, 11b-Tx buffer#3) Rd – [17:16]: The active area of Rx buffer that is operating. (00b-Rx buffer#0, 01b-Rx buffer#1, 10b-Rx buffer#2, 11b-Rx buffer#3)
BA+0x0410	Total transmit length of AxiDMACtrl128 (Low) DG_DMA_LOW_TOTAL_TRANSMIT_LENGTH_OFFSET
	Wr/Rd – [31:0]: 32 lower bits of Total amount of Tx data in 128-bit unit. Valid range is 1-0xFFFFFFFF.
BA+0x0414	Total transmit length of AxiDMACtrl128 (High) DG_DMA_HIGH_TOTAL_TRANSMIT_LENGTH_OFFSET
	Wr/Rd – [20:0]: 21 upper bits of Total amount of Tx data in 128-bit unit. Valid range is 1-0x1FFFFF.
BA+0x0418	Current transmit length of AxiDMACtrl128 (Low) DG_DMA_LOW_CURRENT_TRANSMIT_LENGTH_OFFSET
	Rd – [31:0]: 32 lower bits of current amount of Tx data in 128-bit unit
BA+0x041C	Current transmit length or AxiDMACtrl128 (High) DG_DMA_HIGH_CURRENT_TRANSMIT_LENGTH_OFFSET
	Rd – [20:0]: 21 upper bits of current amount of Tx data in 128-bit unit
BA+0x0420	Total receive length of AxiDMACtrl128 (Low) DG_DMA_LOW_TOTAL_RECEIVE_LENGTH_OFFSET
	Wr/Rd – [31:0]: 32 lower bits of total amount of Rx data in 128-bit unit. Valid range is 1-0xFFFFFFFF.
BA+0x0424	Total receive length of AxiDMACtrl128 (High) DG_DMA_HIGH_TOTAL_RECEIVE_LENGTH_OFFSET
	Wr/Rd – [20:0]: 21 upper bits of total amount of Rx data in 128-bit unit. Valid range is 1-0x1FFFFF.
BA+0x0428	Current receive length of AxiDMACtrl128 (Low) DG_DMA_LOW_CURRENT_RECEIVE_LENGTH_OFFSET
	Rd – [31:0]: 32 lower bits of current amount of Rx data in 128-bit unit
BA+0x042C	Current receive length of AxiDMACtrl128 (High) DG_DMA_HIGH_CURRENT_RECEIVE_LENGTH_OFFSET
	Rd – [20:0]: 21 upper bits of current amount of Rx data in 128-bit unit

Address Wr/Rd	Register Name (Label in the NVMeTCP25DMATest.cpp) Description
BA+0x0400 – BA+0x07FF: AxiDMACtrl128 Control and Status (Write/Read access) ***Note: BA+0x0600 – BA+0x06FF: Tx buffer parameters [CPU -> Hardware]*** ***BA+0x0700 – BA+0x07FF: Rx buffer parameters [Hardware -> CPU]***	
BA+0x0440	Tx buffer status of AxiDMACtrl128 DG_DMA_TXBUFFER_VALID_OFFSET
	Wr/Rd – [3:0]: Each bit is mapped to be the status of each Tx buffer area. Bit[0], [1], [2], and [3] are the status of Tx buffer#0, #1, #2, and #3, respectively. Wr: Set to 1b when the data in Tx buffer#i (where i is the area of Tx buffer) is ready for Tx transfer. Rd: 0b-Data is available in Tx buffer#i, 1b-Data is available in Tx buffer#i. In Tx transfer, this flag is asserted by CPU after finishing preparing the data for each Tx buffer area. It is de-asserted by the hardware kernel after all data is completely read.
BA+0x0444	Rx buffer valid of AxiDMACtrl128 DG_DMA_RXBUFFER_VALID_OFFSET
	Wr/Rd – [3:0]: Each bit is mapped to be the status of each Rx buffer area. Bit[0], [1], [2], and [3] are the status of Rx buffer#0, #1, #2, and #3, respectively. Wr: Set to 1b by CPU to indicate that Rx buffer#i (where i is the area of Rx buffer) is empty for Rx transfer. Rd: 0b-Data is not available in Rx buffer#i, 1b-Data is available for reading in Rx buffer#i. In Rx transfer, this flag is asserted by the hardware kernel after the data is completely prepared. It is de-asserted by CPU after all data is completely read.
BA+0x0480	Buffer size of AxiDMACtrl128 DG_DMA_BUFFER_SIZE_OFFSET
	Wr/Rd – [31:0]: Mapped to the buffer size in byte unit. Maximum size is 4GB. Data bus size of DMA engine is 128 bits, so bit[5:0] of this register must be equal to 000000b. *Note: The hardware kernel loads this register when the reset flag (DG_DMA_RESET_OFFSET) is asserted*.
BA+0x0600 – BA+0x060F: Tx buffer#0 parameters, BA+0x0610 – BA+0x061F: Tx buffer#1 parameters, BA+0x0620 – BA+0x062F: Tx buffer#2 parameters, BA+0x0630 – BA+0x063F: Tx buffer#3 parameters	
BA+0x0600	Tx buffer#0 base address for AxiDMACtrl128 (Low) DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(0)
	Wr/Rd - [31:0]: Mapped to the 32 lower bits of Tx buffer#0 address in the host memory. Value is loaded when AxiDMACtrl128 is reset.
BA+0x0604	Tx buffer#0 base address for AxiDMACtrl128 (High) DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(0)
	Wr/Rd - [31:0]: Mapped to the 32 upper bits of Tx buffer#0 address in the host memory. Value is loaded when AxiDMACtrl128 is reset.
BA+0x0610 - BA+0x061F	Tx buffer#1 parameters for AxiDMACtrl128 0x0610: DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(1) 0x0614: DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(1)
BA+0x0620 - BA+0x062F	Tx buffer#2 parameters for AxiDMACtrl128 0x0620: DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(2) 0x0624: DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(2)
BA+0x0630 - BA+0x063F	Tx buffer#3 parameters for AxiDMACtrl128 0x0630: DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(3) 0x0634: DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(3)

Address Wr/Rd	Register Name (Label in the NVMeTCP25DMATest.cpp) Description
BA+0x0400 – BA+0x07FF: AxiDMACtrl128 Control and Status (Write/Read access) ***Note: BA+0x0600 – BA+0x06FF: Tx buffer parameters [CPU -> Hardware]*** ***BA+0x0700 – BA+0x07FF: Rx buffer parameters [Hardware -> CPU]***	
BA+0x0700 – BA+0x070F: Rx buffer#0 parameters, BA+0x0710 – BA+0x071F: Rx buffer#1 parameters, BA+0x0720 – BA+0x072F: Rx buffer#2 parameters, BA+0x0730 – BA+0x073F: Rx buffer#3 parameters	
BA+0x0700	Rx buffer#0 base address for AxiDMACtrl128 (Low) DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(0)
	Wr/Rd - [31:0]: Mapped to the 32 lower bits of Rx buffer#0 in the host memory Value is loaded when AxiDMACtrl128 is reset.
BA+0x0704	Rx buffer#0 base address for AxiDMACtrl128 (Low) DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(0)
	Wr/Rd - [31:0]: Mapped to the 32 upper bits of Rx buffer#0 in the host memory. Value is loaded when AxiDMACtrl128 is reset.
BA+0x0710 - BA+0x071F	Rx buffer#1 parameters for AxiDMACtrl128 0x0710: DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(1) 0x0714: DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(1)
BA+0x0720 - BA+0x072F	Rx buffer#2 parameters for AxiDMACtrl128 0x0720: DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(2) 0x0724: DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(2)
BA+0x0730 - BA+0x073F	Rx buffer#3 parameters for AxiDMACtrl128 0x0730: DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(3) 0x0734: DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(3)
BA+0x0800 – BA+0x080F: IP information (Read access)	
BA+0x0800	NVMeTCP25G IP Version DG_IPVERSION_INTREG_OFFSET
	Rd – [31:0]: Mapped to IP Version of NVMeTCP25G-IP
BA+0x0804	DGEMAC IP Version DG_EMACVERSION_INTREG_OFFSET
	Rd – [31:0]: Mapped to IP Version of DG EMAC-IP if it is included in the design.

3 The host software

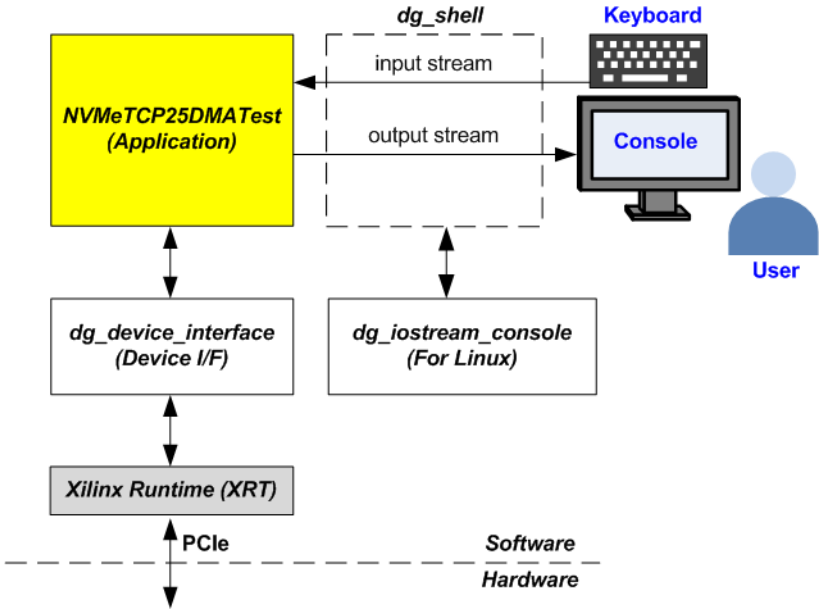

Figure 3-1 The software architecture in NVMeTCP25G-IP on Alveo card demo

The demo's host software comprises two software categories: the application called NVMeTCP25DMATest, and three frameworks named dg_shell, dg_ipstream_console, and dg_device_interface. The dg_shell framework controls the user input (keyboard) and output console (monitor) through dg_iostream_console, which is specially designed for LinuxOS to handle input and output streams using its own control sequence. The dg_device_interface framework is responsible for managing the hardware interface on the Alveo card using Xilinx runtime (XRT), which allows it to write/read hardware registers, manage the device, and handle memory allocation processes.

The Xilinx Runtime Library (XRT) serve as the software interface for communication between the application and the hardware kernels. It uses the PCIe interface to connect the hardware on the Alveo card to the host system. More information on the Xilinx Runtime Library can be found at the following link.
https://www.xilinx.com/products/design-tools/vitis/xrt.html

Further details of the software used in the demo are provided below.

3.1 Framework

This topic describes two software frameworks - the device interface and the shell. The device interface framework utilizes the Xilinx Runtime (XRT) to interface with the hardware kernels through a simple function. While the shell framework manages the input and output of the console (Linux terminal) to provide a user interface.

3.1.1 Device interface

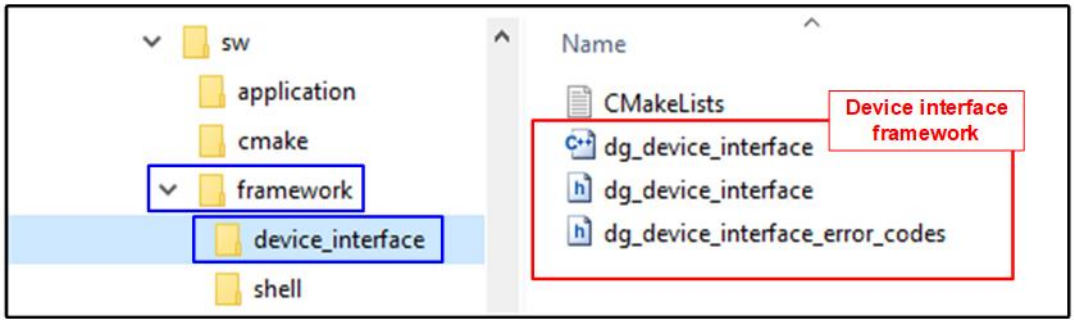

Figure 3-2 Device interface framework

The application uses the device interface to communicate with the hardware kernels via the Xilinx Runtime Library (XRT). It is responsible for establishing a connection, allocating buffers, and writing/reading hardware registers. The device_interface directory contains three source codes, as shown in Figure 3-2.

- "dg_device_interface_error_codes.h": Defines the returned value of the function for the device interface, which can be "OK" status or error codes. These definitions are referenced by some functions in the class. The following values are listed.
  - DG_OK
  - DG_DEV_INTERFACE_ERROR_FAIL_TO_ASSOCIATE_WITH_XRT
  - DG_DEV_INTERFACE_ERROR_DEVICE_IS_NOT_OPENED
  - DG_DEV_INTERFACE_ERROR_CANNOT_OPEN_DEVICE
  - DG_DEV_INTERFACE_ERROR_CU_NAME_NOT_FOUND
  - DG_DEV_INTERFACE_ERROR_HOST_MEMORY_INTERFACE_NOT_FOUND
  - DG_DEV_INTERFACE_ERROR_IP_KERNEL_AND_HOST_MEMORY_MISMATCH
  - DG_DEV_INTERFACE_ERROR_FAIL_ALLOCATEBUFFER

- "dg_device_interface.h": Declares a class and functions defined in C++ source file (dg_device_interface.cpp). The header file determines which function can be called from other classes and declares variables in the class with an initial value if specified.

- "dg_device_interface.cpp": Contains the general functions for connecting the device, accessing hardware register, and allocating/de-allocating host memory. The device interface framework's function lists are described as follows.
  Note: The compute unit name string (cu_name) is defined as a constant in the software source code – "NVMeTCP25DMATest: NVMeTCP25DMATest". This value must be updated if the user changes the hardware kernel name.

Device Connection

uint32_t CreateDeviceIF(void)	
Parameters	None
Return value	DG_OK: Success, Error code: Error found
Description	Use an XRT function to connect with the hardware platform to retrieve an information such as the CU (Compute Unit) address. The CU address is applied for writing/reading the hardware register. DG_OK is returned when CU name is correct as mentioned in the above note. Otherwise, Error code is returned.

char* GetDeviceName(void)	
Parameters	None
Return value	Character pointer of the device name
Description	Return a device name that is obtained while initializing the device interface from CreateDeviceIF function.

void Close(void)	
Parameters	None
Return value	None
Description	Use an XRT function to disconnect the software application from the hardware if the connection is created.

Write/Read Register

uint32_t ReadIntReg(uint64_t offset)	
Parameters	offset: The address offset of the hardware register to be read
Return value	Read value from the hardware register
Description	Calculate the actual address by adding the CU address with the input offset. Next, use an XRT function and the actual address to read the data in the hardware register. Finally, return the read data back to user.

void WriteIntReg(uint64_t offset, uint32_t value)	
Parameters	offset: The address offset of the hardware register to be written value: 32-bit unsigned value for writing to the register
Return value	None
Description	Calculate the actual address by adding the CU address with the input offset. Next, use an XRT function and the actual address to write the input value into the hardware register.

Buffer Management

uint32_t AllocateBuffer(uint32_t sizeInBytes, void*& HostAddr, uint64_t* HWAddr)	
Parameters	sizeInBytes: The memory size for allocating in byte unit HostAddr: The pointer of the virtual host base address HWAddr: The pointer to the hardware base address.
Return value	DG_OK: Success, Error code: Error found
Description	Use an XRT function to retrieve an information of the hardware and then use this information to verify the connection between the hardware kernel and the host memory. After that, allocate the host memory via the XRT function (the memory size is set by the input). Finally, update the virtual host base address, hardware base address, and the local variables of the host memory details.

void FreeBufferHostOnly(void)	
Parameters	None
Return value	None
Description	If the host memory is allocated, use an XRT function to free the host memory and clear the local variables of the host memory details.

3.1.2 Shell

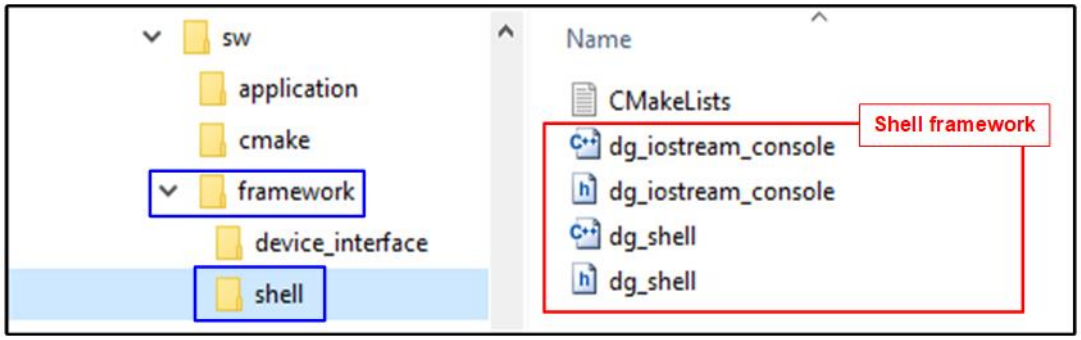

Figure 3-3 Shell framework

The shell framework, named dg_shell, manages the input and output stream on the Linux terminal (console). It receives keyboard input, handles the input string, parses the input data type, and prints strings to the console. To work with the Linux terminal, the shell framework uses an I/O stream console library (dg_iostream_console) for input and output stream console management such as altering the terminal environment, getting user keyboard input, and pushing the printed string output to terminal.

There are two source codes for handling input and output stream console as shown in Figure 3-3.

- "dg_iostream_console.h": This header file declares a class and functions that are defined in the C++ source file (dg_iostream_console.cpp). It specifies which function can be called from other classes and declares variables in the class with ab initial value if specified.

- "dg_iostream_console.cpp": This source code defines the general function for managing the input and output stream on the Linux terminal environment, including writing strings to the console, changing the terminal configuration to be used by the shell, reverting the terminal configuration to the original one, and getting the input character from the user through the terminal.

  The function lists of the I/O stream classes are described below.

  Note:
  - *When constructing the "InStreamConsole" object, it requires the pointer of the output stream object.*
  - *"KeyPressEnum" is a C++ enumeration declared in the header file (dg_iostream_console.h) containing the keyboard input type for processing in the shell framework, including NORMAL, BACKSPACE, LEFTARROW, RIGHTARROW, DELETE, TAB, EOL, and CONTROL.*

OutStreamConsole class

void write(const char* s, uint32_t numChars)	
Parameters	s: Pointer to the character for printing out on the console numChars: The character length of “s”
Return value	None
Description	Call function (fwrite) to write the output (stdout) by the character “s” which specifies the length from “numChars”. Next, flush the output to the terminal.

void erase(uint32_t numChars)	
Parameters	numChars: The number of characters to delete from the terminal
Return value	None
Description	Delete the currently displayed character on the terminal which specifies the length from “numChars”.

InStreamConsole class

void NewSetting(void)	
Parameters	None
Return value	None
Description	Change the Linux terminal configuration to non-echo mode and to process an input from the terminal without endline character. The initial configuration is saved in a local variable for future retrieval.

void RestoreSetting(void)	
Parameters	None
Return value	None
Description	Restore the Linux terminal configuration to the initial configuration by using a local variable.

void getChar(char* pChar)	
Parameters	pChar: Pointer to store the input character
Return value	None
Description	Get a character input from the Linux terminal and write to the pointer. When this function is called, it waits until an input is received.

void FlushInputStream(void)	
Parameters	None
Return value	None
Description	Flush the input stream for the Linux terminal. This function is recommended to use before using “getChar” function.

int GetInputCharLength(void)	
Parameters	None
Return value	Number of input characters in the Linux terminal buffer
Description	Determine the number of user input characters in the Linux terminal buffer.

KeyPressEnum getKeyPress(char c)	
Parameters	c: Character input to determine the character type.
Return value	NORMAL: General character that can be printed BACKSPACE, RIGHTARROW, LEFTARROW, DEL, TAB, EOL: Special characters that have specific operation CONTROL: Control character that does not have the operation
Description	Determine the type of the input character and returns the value.

The shell framework consists of two source codes, outlined as follows.

- "dg_shell.h": Similar to the "dg_iostream_console.h" header file, this header file declares a class, functions, and variables that are defined in the corresponding C++ source file (dg_shell.cpp).

- "dg_shell.cpp": This source file provides the main function of the shell framework, including simplify the management of input and output on the console and offering utility functions such as string parsing to an unsigned integer. The following is a list of functions included in the shell framework.

General Function

void Initialise(DG::InStreamConsole& inputStream, DG::OutStreamConsole& outputStream)	
Parameters	inputStream: The input stream object outputStream: The output stream object
Return value	None
Description	Load the pointers of the input stream object and the output stream output to the local variables for using in the shell framework.

void ClearInputBuffer(void)	
Parameters	None
Return value	None
Description	Clear the input buffer which is the internal variable.

bool GetInputLine(char*& pStr)	
Parameters	pStr: The reference of the pointer of the input line string
Return value	True: The operation is successful. False: Fail to retrieve an input character using "getChar" function.
Description	1. Clear the input buffer and the input stream using the "ClearInputBuffer" and "FlushInputStream" functions. 2. Receive input from the terminal using the "getChar" function. 3. Process the input using the "ProcessInputChar" function. 4. Repeat step 2-3 until the end-of-line is detected. 5. Finally, set the pointer to the first character of the input buffer.

bool FlushInputBuffer(void)	
Parameters	None
Return value	True: The operation is successful (The function now returns only “True”).
Description	Flush the input buffer by calling “FlushInputStream”. This function maps the function of “dg_iostream_console” for using in the application.

bool IsAnyInputKey(void)	
Parameters	None
Return value	True: Some inputs are received from the Linux terminal. False: No received input from the terminal
Description	Read the number of received input from the terminal using “GetInputCharLength” function. If the length is greater than zero, Return “True” as a result. Otherwise, return “False”.

int printf(const char* fmt, ...)	
Parameters	fmt: String that contains the text to be printed on the console arguments: Additional arguments
Return value	Number of input characters of the output buffer
Description	Used to receive the input string with the arguments and then calculate the length for writing to the output stream. It is almost similar to standard “printf” function, but it sends to the output stream for displaying on the console.

Input Parsing Function

bool parseUInt32(char* pInputstr, uint32_t* pValue)	
Parameters	pInputstr: Pointer to the input string for processing pValue: Pointer of 32-bit result after parsing
Return value	True: The operation is successful False: Fail to parse the input or other errors
Description	Convert the input string of the decimal value to be 32-bit unsigned value. The input range must not be more than FFFF_FFFFh.

bool parseHex32(char* pInputstr, uint32_t* pValue)	
Parameters	pInputstr: Pointer to the input string for processing pValue: Pointer of 32-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Convert the input string of the hex value to be 32-bit unsigned value. The input range must not be more than FFFF_FFFFh.

bool parseUInt64(char* pInputstr, uint64_t* pValue)	
Parameters	pInputstr: Pointer to the input string for processing pValue: Pointer of 64-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Convert the input string of the decimal value to be 64-bit unsigned value.

bool parseHex64(char* pInputstr, uint64_t* pValue)	
Parameters	pInputstr: Pointer to the input string for processing pValue: Pointer of 64-bit result after parsing
Return value	True: the operation is successful False: Fail to parse the input string or other errors
Description	Convert the input string of the hex value to be 64-bit unsigned value.

bool get_input_long(uint64_t* pValue)	
Parameters	pValue: Pointer of 64-bit result of the input from the terminal
Return value	True: The operation is successful False: Fail to retrieve an input character, to parse the input string, or other errors
Description	Call “GetInputLine” function to retrieve a string input and then call “parseUInt64” or “parseHex64” function to parse the input, depending on the input format. Finally, return the result after parsing.

bool get_ipv4_addr(uint32_t* pValue)	
Parameters	pValue: Pointer to return 32-bit IPv4 address value that is received from the terminal
Return value	True: The operation is successful False: Fail to retrieve an input character, to parse the input string, or other errors
Description	Call “GetInputLine” function to retrieve a string input and then verify the input format. Error is returned if the input is invalid. Otherwise, the input is converted to 32-bit unsigned value to be the returned result.

Input Key Processing Function

void ProcessInputChar(char c)	
Parameters	c: Character input
Return value	None
Description	Use “keyPressEnum” (function of InStreamConsole) to determine the character type. The function that is called for processing depends on the character type.

void ProcessNormalChar(char c)	
Parameters	c: Character input
Return value	None
Description	Add the new character to the buffer and then print to the output stream.

void ProcessBackspace(void)	
Parameters	None
Return value	None
Description	Delete the left side character and then print to the output stream.

void ProcessEOL(void)	
Parameters	None
Return value	None
Description	Add NULL to the buffer and then print to the output stream. After that, set the local variable that shows the end of line flag to be “true”. When “GetInputLine” function detects the end of line flag, the input buffer will be cleared.

void ProcessTab(void)	
Parameters	None
Return value	None
Description	Move the console cursor to the end of line input string.

void ProcessLeftArrow(void)	
Parameters	None
Return value	None
Description	Move the console cursor to the left side for one position.

void ProcessRightArrow(void)	
Parameters	None
Return value	None
Description	Move the console cursor to the right side for one position.

void ProcessDel(void)	
Parameters	None
Return value	None
Description	Delete the character at the current position of the console and then print to the output stream.

3.2 Application

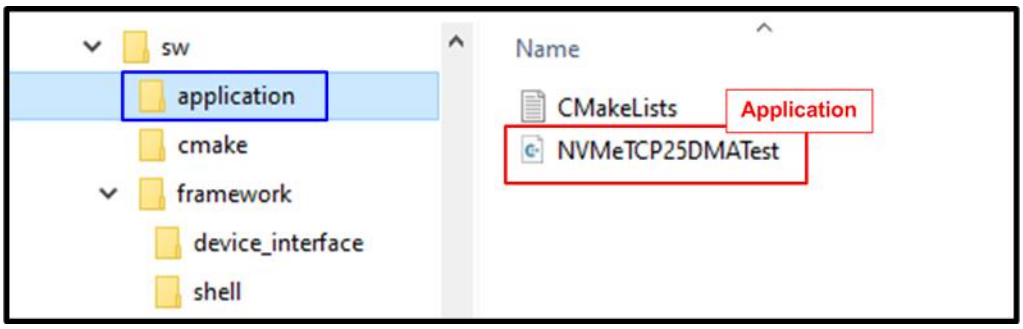

Figure 3-4 Application layer

Figure 3-4 shows the file location of the application source code, NVMeTCP25DMATest.cpp. The steps of the operation in the main function are as follows.

1) Start a signal handler to detect user input of “CTRL+C”. If the key is detected, the termination process will be initiated, which includes the following steps.
   i) Turn off the connection for all NVMeTCP25G-IPs in the hardware kernel.
   ii) Reset the DMA engine in the hardware kernel.
   iii) De-allocate the memory using XRT.
   iv) Close the device interface.
   v) Restore the terminal configuration to the initial configuration.
2) Start a signal thread handler to detect user signal (“SIGUSR1”) which will be used in a future test sequence for handling multi-threads.
3) Establish a connection to the hardware platform using the device interface framework. Configure the system to interface with the hardware kernel and the Linux terminal.
4) Check for the availability of NVMeTCP25G-IP in the hardware kernel and display the IP information.
5) Allocate 1 GB memory in the host system through the device interface to prepare for the DMA feature.
6) Set up input/output stream and the Linux terminal for the application using the shell object.
7) The DMA environment for both the hardware platform and software application is initialized in the following steps.
   i) Assign a variable to point to the allocated memory from device interface. Two pointer types are used - the host virtual memory address pointer in the software and the hardware memory address used by the hardware. Both pointers point to the same memory location.
   ii) Send a soft reset to the DMA logic in all channels in the hardware platform (DG_DMA_RESET_OFFSET[i][0]=1b where “i” is the selected channel).
   iii) Set the buffer size in the hardware register (DG_DMA_BUFFER_SIZE_OFFSET[i] where “i” is the selected channel) and then set the 64-bit hardware memory address of the transmit and receive buffers to the hardware registers. In this reference design, either transmit operation or receive operation is performed at a time, so both the transmit and receive buffers are utilized and set to the same memory block. The whole allocated memory is divided into four areas for each hardware channel, and each channel memory is further split into four areas for multi-buffering, equally. Therefore, the total allocated memory is divided into sixteen blocks, and each channel has four blocks of the distributed memory.

Note: DG_DMA_TX/RXBUFFER_LOW/HIGH_ADDRESS_OFFSET[i][j] are used to set the 64-bit hardware memory address of the transmit/receive buffer where "i" is the selected channel and "j" is the buffer position of the selected channel.

iv) De-assert the soft reset to the DMA logic in all channels (DG_DMA_RESET_OFFSET[i][0]=0b where "i" is the selected channel).

v) The host virtual memory address pointer is calculated in the same way as the hardware memory address for later use in the operation.

8) Once all initialization processes have been completed, the application's main menu will be displayed on the console, as shown in Figure 3-5. The menu offers three test operations that the user can choose: Set parameter and connect, Write/Read command, and Disconnect. The user must select the appropriate menu based on the following criteria.

i) Set network parameters and connect to the target SSD. This menu is the first menu that user interacts with after starting the software application. If all channels are already connected to the target SSDs, this menu cannot be accessed.

ii) After some channels are connected to the target SSDs, the user can write or read the target SSDs using the Write/Read command.

iii) If the user wishes to disconnect from the target SSD (disconnect NVMeTCP25G-IP from the target), they can select the Disconnect option.
Note: After disconnection, the user can change parameters such as network parameters or NVMe parameters to match the new target. If the user reconnects the host to the same target, they can skip setting new parameters.

The main menu only displays the appropriate menu options. If the user selects a menu that is not displayed on the console, no action will be taken. Further details of each menu are described below.

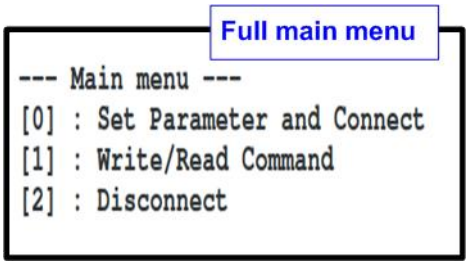

Figure 3-5 System initialization in Client mode by using default parameters

3.2.1 Set parameter and connect

This menu is used to set the parameters to an NVMeTCP25G-IP for connecting to the target SSD and then the connection establishment is initiated. The software application will display the status table of all channels and ask the user to select a channel for the operation. If the selected channel is already connected or has no connectivity due to Ethernet link being down, the operation will be cancelled.

In this menu, the user can set the necessary parameters for NVMeTCP25G-IP initialization, which are host MAC address, host IP address, host port number, Target MAC address mode, Target MAC address (when running Fixed MAC mode), and Target IP address. The NVMe Qualified Name (NQN) of the host and target also need to be set. The parameter setting and connection process follows these steps.

1) The software application displays the current parameter values on the console. If the parameters have never been set, default values assigned in the software application are shown.
2) The user can skip the parameter setting (use the current values) by pressing ‘x’ on the keyboard or set the desired values by pressing other keys.
3) If the user presses a key other than ‘x’, they are asked to input the desired parameter values which are Target NQN, host MAC address, host IP address, host port numbers (Admin and I/O), Target MAC address mode, Target MAC address (when running Fixed MAC mode), and Target IP address, respectively. Invalid inputs will not update the parameters and will use the latest value.
4) Once the parameter values are updated (or ‘x’ key is pressed), the software application sets them to the parameter registers in the hardware of the selected channel. However, some parameters of NVMeTCP25G-IP cannot be set by the user but use the predefined values in the software application. Any changes to these values would require modification to the software application. The following parameter values are defined by the application.
   - TCP timeout value: 1 sec
   - NVMe timeout value: 4 sec
   - NQN of the host value of channel#0-#3: “dgnvmehtest0”, “dgnvmehtest1”, “dgnvmehtest2”, and “dgnvmehtest3”, respectively.
5) After setting the parameter values, the software application enables the connection for the selected NVMeTCP25G-IP by setting DG_NVMETCP_HCONNSTS_INTREG_ OFFSET[i][0]=1b where “i” is the selected channel.
6) The application then waits for the connection to be fully established by monitoring the host connection status flag of the selected channel (DG_NVMETCP_HCONNSTS_ INTREG_OFFSET[i][1]=1b where “i” is the selected channel). If any errors are found during this process (DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][3]=1b where “i” is to the selected channel), the application stops and displays an error message.
7) Once the connection is established, a message indicating successful connection (“Connect target successfully”) is displayed. Additionally, the status table of all channels is shown, which includes the connection status and the target SSD capacity in GB unit.

3.2.2 Write/Read command

In this menu, the NVMeTCP25G-IP with DMA feature can be tested by two types of operations: Write command operation and Read command operation.

Write command operation
The Write command operation involves generating data in the host software application, filling the data into the host memory (Tx buffer: Transmit buffer), transferring the data to hardware, and writing the data to the target SSD using NVMeTCP25G-IP. Before the test begins, a child thread is created to write the pattern data, such as incremental, decremental, all zero's, or all one's pattern, to the Tx buffer. The main thread then sets the control flag to the hardware register for reading the data from the Tx buffer and transferring it to the NVMeTCP25G-IP as a write data command to be stored in the target SSD. The operation is considered complete when all data is written from the host system to the target SSD.

To handle the flow control of the Tx buffer, three counters are used: the write counter of Tx buffer, which is increased by the child thread after finishing writing data of each Tx buffer area; the request counter, which is increased by the main thread after sending the request to the hardware to start reading the new data from each Tx buffer area; and the read counter of the Tx buffer, which is increased by the main thread when the hardware sets the clear flag after finishing reading the data of each Tx buffer area.

Read command operation
The Read command operation involves reading data in the target SSD through NVMeTCP25G-IP, storing the data in the host memory (Rx buffer: Receive buffer) from hardware using DMA, and processing the data by the host software application. Before the test begins, a child thread is created to read data from the Rx buffer, which is written by the hardware. The received data is verified depending on the test pattern input, such as incremental, decremental, all zero's, or all one's pattern. The operation is considered complete when all data is received and the connection is closed by the target.

Similar to the Tx buffer, three counters are used to handle the flow control of the Rx buffer: the write counter of the Rx buffer, which is increased by the main thread when the hardware sets the new valid flag after finishing writing data to each Rx buffer area; the read counter of the Rx buffer, which is increased by the child thread after finishing reading data of each Rx buffer area; and the complete counter, which is increased by the main thread after setting the clear flag to each Rx buffer area.

In summary, the user selects the operation of each channel individually which is Write command operation, Read command operation, or no operation. Next, the user sets test parameters for Write/Read command operation (start address, transfer length, and test pattern). The operation is cancelled if some inputs are invalid. The Write command operation generates data in the host software application and writes it to the target SSD using NVMeTCP25G-IP, while the Read command operation reads data from the target SSD and stores it in the receive buffer for processing by the host software application.

Note: The NVMeTCP25G-IP fixes the data size of each Write/Read command to 8 Kbytes, so the start address and transfer length that are input by the user on the console which are 512-byte unit must be aligned to 16.

To operate the Write/Read command, follow these steps. Note that some register names have the index [i] to refer to the channel, which can be 0-3 for four connections.

1) Display the channel number, host NQN name, and target NQN of the channel.
2) Receive test parameters for the channel from the user (start address, transfer length, test pattern, and test mode which is Write/Read/No operation) and verify if all inputs are valid as described in the menu description.
3) Repeat step 1) and 2) to get the parameters of the next active channel until the current channel is the final active channel.
4) Set up the hardware for transferring data between the host memory and the NVMeTCP25G-IP. Calculate the following parameters using the received test parameters.
   i) Set the start address to DG_NVMEIF_LOW/HIGH_START_ADDRESS_OFFSET[i]
   ii) Set the total transfer length to DG_NVMEIF_LOW/HIGH_TOTAL_TRANSFER_LENGTH_OFFSET[i].
   iii) Set the operation mode to DG_NVMEIF_COMMAND_OFFSET[i][0]. If the user selected Write command operation for this channel, set this register to 1b. Otherwise, set it to 0b for Read command operation.
   iv) Start the operation by setting DG_NVMEIF_CONTROL_OFFSET[i][0] to1b
5) Prepare the parameters and software application for the DMA feature.
   i) Calculate the amount of data for the last Tx/Rx buffer area and the total count of Tx/Rx buffer areas required to store all data.
   ii) For Write command operation, create a child thread called gen_txbuf_data to write the data into the Tx buffer until it is completely filled. For Read command operation, create a child thread called ver_rxbuf_data to read the data from the Rx buffer until it is completely read.
6) Setup the hardware for DMA function.

   For Write command operation, set the total transmit length to DG_DMA_TOTAL_TRANSMIT_LENGTH_OFFSET[i] and start the Tx DMA engine hardware by setting DG_DMA_COMMAND_OFFSET[i][0] to 1b.

   For Read command operation, set the total receive length to DG_DMA_TOTAL_RECEIVE_LENGTH_OFFSET[i] and start the Rx DMA engine hardware by setting DG_DMA_COMMAND_OFFSET[i][1] to 1b.

7) Control the operation of the NVMeTCP25G-IP and Tx/Rx DMA engine by running the following steps in a forever loop until the termination condition of all channels is met.

Termination condition

i) Check that the IP operation is completed from the busy status (DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][2]=0b) by checking the busy of the IP and the last buffer of the DMA operation is completely transferred (Write command operation: read counter = last one and Read command operation: complete counter = last one).
ii) Ensure that all data in the buffer of the DMA operation is completely transferred.
   a. For Write command operation: Read counter of Rx buffer = set value.
   b. For Read command operation: Complete counter = set value.

Write command operation

i) Check for errors in NVMeTCP25G-IP. If an error is detected (DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][3]=1b), terminate the child thread and change the control operation of this channel to “no operation”.
ii) Check if there is new data filled by the child thread (request counter < write counter). If not, proceed to the next step. Otherwise, set the valid flag of the new Tx buffer area to the hardware (DG_DMA_TXBUFFER_VALID_OFFSET[i][j]=1b where ‘j’ is an index of the new buffer area) and increase the request counter.
iii) Check if the new clear flag of Tx buffer is returned by the hardware after finishing reading the data from Tx buffer (DG_DMA_TXBUFFER_VALID_OFFSET[i][j]=0b where ‘j’ is an index of the new buffer area). If not, proceed to the next step. Otherwise, increase the read counter.

Read command operation

i) Check for errors in NVMeTCP25G-IP. If an error is detected (DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][3]=1b), terminate the child thread and change the control operation of this channel to “no operation”.
ii) Check if there is a new valid flag of Rx buffer (DG_DMA_RXBUFFER_VALID_OFFSET[i][j]=1b where ‘j’ is an index of the new buffer area) that is set by the hardware after finishing writing the data to Rx buffer and Rx buffer is still not full. If not, proceed to the next step. Otherwise, increase the write counter.
iii) When the child thread detects the updated write counter, it starts reading and verifying the data from Rx buffer (if the verification flag is set). The read counter is updated by the thread after finishing the operation.
iv) Check if the read counter is updated by the child thread (read counter > complete counter). If not, proceed to the next step. Otherwise, set clear flag of the new Rx buffer area to the hardware (DG_DMA_RXBUFFER_VALID_OFFSET[i][j]=1b where ‘j’ is an index of the new buffer area). After that, increase the complete counter by the main thread.

Display the progress of all channels every second. Calculate the total amount of transmitted data and received data by the software application and display it on the console.

8) Wait until the child thread (gen_txbuf_data or ver_rxbuf_data) finishes the operation.
9) For the channel that has run the read command operation, display error messages if errors have occurred (Expected data and received data are mismatched).
10) Calculate performance and display test result on the console.

3.2.3 Disconnect

This menu allows the user to terminate the TCP/IP and NVMe/TCP connections between the host and the target of a specific channel that were established using the Connect command. The software application displays the status table of all channels and prompts the user to select the channel to run the Disconnection operation. If an unconnected channel is selected, the operation is cancelled. The sequence of Disconnect is as follows.

1) The connection enable of the NVMeTCP25G-IP in the selected channel is reset by setting DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][0]=0b where “i” is the selected channel.
2) The application waits until the disconnection process is completed by monitoring the host connection status flag of the selected channel (DG_NVMETCP_HCONNSTS_INTREG _OFFSET[i][1]=0b where “i” is the selected channel). If any errors are found (DG_NVMETCP_HCONNSTS_INTREG_OFFSET[i][3]=1b where “i” is the selected channel), the process stops with an error message displayed.
3) Finally, if the operation is successful, a message (“Disconnect target successfully”) is displayed, and the status table of all channels, including their connection status, is shown.

3.2.4 Function list in application

This topic describes the function list to run NVMeTCP25G-IP with DMA operation on the Alveo card.

Timer Utilization Function

static inline bool IsTimerRunning(uint32_t index)	
Parameters	index: An index to a timer that is to be used or measured
Return value	TRUE: The timer is running FALSE: The timer is not running
Description	Check whether the timer is running or not.

static void TimerStart(uint32_t index)	
Parameters	index: An index to a timer that is to be used or measured
Return value	None
Description	Start the specified timer and save the starting time to be used later in computing the elapsed time.

static void TimerStop(uint32_t index)	
Parameters	index: An index to a timer that is to be used or measured
Return value	None
Description	Stop the specified timer and save the finishing time to be used later in computing the elapsed time.

template<typename dur_unit> static double TimerElapsed(uint32_t index)	
Parameters	index: An index to a timer that is to be used or measured
Return value	The elapsed time value
Description	Check if the specified timer is currently run. If the timer has been stopped, return the total time usage measured from calling the TimerStart function to calling the TimerStop function. On the other hand, if the timer is still running, return the time that has passed since the timer was started. *Note: The time unit can be specified when using the function. For example, uses "timer.elapsed<std::chrono::seconds>()" to get the elapsed time in seconds.*

Termination Function

void cleanup(void)	
Parameters	None
Return value	None
Description	This function is called for a safety termination of the application. It resets the hardware system by setting DG_DMA_RESET_OFFSET to 1b and sets DG_NVMETCP_HCONNSTS_INTREG_OFFSET to 0b to disconnect from target SSD. Next, de-allocate the memory back to the host through XRT. After that, close the target device interface. Finally, restore the terminal configuration using the initial configuration.

void sigintHandler(void)	
Parameters	None
Return value	None
Description	This function is run to terminate the application when user inputs “CTRL+C”. “cleanup” function is called and then the application is exited.

Console Display Function

static char* code_to_string(uint32_t code)	
Parameters	code: Input value returned from the device interface function
Return value	Pointer to the string of code after conversion
Description	Convert the unsigned value to the string. The error code is defined in the Device interface framework.

const char* convertSizeToString(uint64_t size_input, char* pOutString)	
Parameters	size_input: Transfer size in byte unit to be converted pOutString: Pointer to an array character to store the result
Return value	Pointer to the result string
Description	Store a transfer size input as a string in the buffer associated with pOutString. The converted transfer size is in either MB, GB, or TB unit.

void ShowTestResult(uint64_t testsize, double time_val)	
Parameters	testsize: Transfer size in byte unit to be calculated and displayed time_val: Total time usage in millisecond unit
Return value	None
Description	Display total amount of the input transfer size and total time usage in either seconds, milliseconds, or microseconds. Finally, calculate the performance and display it in MB/s unit.

const char* convertIPAddressToString(uint32_t ip_addr, char* pOutString)	
Parameters	ip_addr: IPv4 address in 32-bit unsigned integer format to be converted pOutString: Pointer to an array character to store the result
Return value	Pointer to the result string
Description	Get IPv4 address in decimal unit, separate it by dot character, store to the buffer associated with pOutString.

void ShowCurrentParameter(uint32_t num)	
Parameters	num: An index of NVMeTCP25G-IP
Return value	None
Description	Read the latest value of the specified NVMeTCP25G-IP's parameters and then display them on the console such as NQN, IP address, MAC address, and port number.

void ShowTestpin(uint32_t num)	
Parameters	num: An index of NVMeTCP25G-IP
Return value	None
Description	Read DG_NVMETCP_NVMTESTPIN_INTREG_OFFSET[num][0-3] to display IP test pin on the console for debugging.

void ShowError(uint32_t num)	
Parameters	num: An index of NVMeTCP25G-IP
Return value	None
Description	Read DG_NVMETCP_HERRTYPE_INTREG_OFFSET[num], decode the error flag, and display the error message associated with that flag.

void DisplayAllChannelStatus(void)	
Parameters	None
Return value	None
Description	Display the table that shows the status of all channels, including three parameters for each channel: Host NQN, Target NQN, and the size of the disk.

void DisplayLineConnectorOfTable(testParameter * pArg[])	
Parameters	pArg: Array pointer to the test parameters of all channels
Return value	None
Description	Draw a line to be a border for the header of the table which is used to show the current test size.

void DisplayHeaderTable(testParameter * pArg[])	
Parameters	pArg: Array pointer to the test parameters of all channels
Return value	None
Description	Display the table header indicating the current test size for all active channels. Also, update the global variable "maxStrOfDisplayTransfer" which will later be used in "DisplayTransferSizeTable" function.

void DisplayTransferSizeTable(testParameter * pArg[])	
Parameters	pArg: Array pointer to the test parameters of all channels
Return value	None
Description	Display the current test size of all active channels in the table.

Thread Function

static void sigThreadHandler(int sig)	
Parameters	sig: signal input that sends to this function to be handled
Return value	None
Description	Call the signal handler to catch the signal for killing the thread next time. Next, kill the thread function where the signal input is corresponded to using “pthread_exit”.

static void kill_thread(std::thread &t)	
Parameters	t: Pointer to a thread ID to be terminated
Return value	None
Description	Use to terminate the thread. Typically, it is invoked when an error has occurred.

static void gen_txbuf_data(const std::atomic<uint32_t> &buf_rdcnt, std::atomic<uint32_t> &buf_wrcnt, volatile uint32_t *head_ptr, const uint32_t buf_totalcnt, const uint32_t last_buf_len, const uint64_t ssd_base_addr, const bool gen_patt)	
Parameters	buf_rdcnt: Pointer to the read counter of Tx buffer buf_wrcnt: Pointer to the write counter of Tx buffer head_ptr: Pointer to the start address of Tx buffer buf_totalcnt: Total count of Tx buffer area that is used in this operation last_buf_len: Buffer size in byte unit of the last buffer in this operation ssd_base_addr: Start address in byte unit of the target SSD gen_patt: 0-incremental, 1-decremental, 2-all zero’s, and 3-all one’s
Return value	None
Description	If “buf_wrcnt” value is greater than or equal to “buf_totalcnt”, the operation is terminated. Before writing new data, the Tx buffer’s status should be checked by reading “buf_wrcnt” and “buf_rdcnt” to ensure it is not already full. Then, calculate the starting address for writing the data and determine the total transfer length for this loop. Fill the Tx buffer with the selected test data pattern, which can be either incremental, decremental, all zeroes, or all ones. Also, the first 8-byte of each 8Kbyte data block is replaced by the header value which is the SSD starting address for storing this 8Kbyte data in 512-byte unit. Finally, increment the “buf_wrcnt” value.

static void ver_rxbuf_data(const std::atomic<uint32_t> &buf_wrcnt, std::atomic<uint32_t> &buf_rdcnt, volatile uint32_t *head_ptr, const uint32_t buf_totalcnt, const uint32_t last_buf_len, const uint64_t ssd_base_addr, const bool ver_patt std::atomic<bool> &ver_fail, std::atomic<uint32_t> &exp_fail, std::atomic<uint32_t> &recv_fail, std::atomic<uint64_t> &addr_fail)	
Parameters	buf_wrcnt: Pointer to the write counter of Rx buffer buf_rdcnt: Pointer to the read counter of Rx buffer head_ptr: Pointer to the start address of Rx buffer buf_totalcnt: Total count of Rx buffer area that is used in this operation last_buf_len: Buffer size in byte unit of the last buffer in this operation ssd_base_addr: Start address in byte unit of the target SSD ver_patt: 0-incremental, 1-decremental, 2-all zero's, and 3-all one's ver_fail: TRUE-verification fail, FALSE-verification successful exp_fail: The first expected 32-bit data that fails to match recv_fail: The first received 32-bit data that fails to match addr_fail: The first 64-bit base address that fails to match *Note: exp_fail, recv_fail, and addr_fail are applied when ver_fail is TRUE.*
Return value	None
Description	If "buf_rdcnt" is greater than or equal to "buf_totalcnt", the operation is terminated. To initiate the new operation, it must be confirmed that new data has been stored in the Rx buffer (buf_rdcnt is less than buf_wrcnt). After that, calculate the starting address to read the data and the total transfer length for this loop. Next, iterate through each 8Kbytes block in the Rx buffer to process the data. "ver_patt" is used to verify the received data, which could be incremental, decremental, all zeroes, or all ones patterns. Similar to "gen_txbuf_data" function, the first 8 bytes of each 8Kbyte data block are verified by the header value, representing the SSD's starting address for storing this 8Kbyte data in 512-byte unit. Finally, increment the "buf_rdcnt" value.

Buffer Handler Function

inline bool check_txbuf_clear(uint32_t txbuf_ctrl, uint32_t index)	
Parameters	txbuf_ctrl: Read value of DG_DMA_TXBUFFER_VALID_OFFSET index: The index of Tx buffer area
Return value	TRUE: This Tx buffer area is free and ready to fill data FALSE: This Tx buffer area is full
Description	Checking whether the specified Tx buffer area is free or not by reading txbuf_ctrl.

inline bool check_rxbuf_valid(uint32_t rxbuf_ctrl, uint32_t index)	
Parameters	rxbuf_ctrl: Read value of DG_DMA_RXBUFFER_VALID_OFFSET index: The index of Rx buffer area
Return value	TRUE: This Rx buffer area has the new data for reading FALSE: This Rx buffer area does not have the new data
Description	Checking whether the specified Rx buffer has data or not by reading rxbuf_ctrl.

Miscellaneous Function

int GetParameterOfChannel(uint32_t channel, testParameter * pArg)	
Parameters	channel: An index of channel to receive test parameters pArg: Pointer to the struct of test parameters of the channel
Return value	0: The operation is successful -1: The operation failed
Description	Get input test parameters of the specified channel from user such as Test mode, Start address, transfer size, and test pattern.

bool IsEthernetLink(uint32_t index)	
Parameters	index: An index of the Ethernet channel
Return value	TRUE: Ethernet connection is up FALSE: Ethernet connection is down
Description	Check Ethernet link status from the Ethernet layer in the hardware channel associated with the specified NVMeTCP25G-IP.

void set_param(uint32_t num)	
Parameters	num: An index of NVMeTCP25G-IP
Return value	None
Description	Set parameters of the specified NVMeTCP25G-IP using the parameters in the software application.

Test Function

int proc_setconnect(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found -2: Fatal error is found (The system needs to be reset)
Description	Run Set parameter and connect following description in topic 3.2.1.

int wrrd_dev(volatile uint32_t * dmaHostPtr[])	
Parameters	dmaHostPtr: Array pointer of the host virtual address (channel buffer)
Return value	0: The operation is successful -1: Fatal error is found (The system needs to be reset)
Description	Run Write/Read command following description in topic 3.2.2.

int proc_disconnect(void)	
Parameters	None
Return value	0: The operation is successful -1: Receive invalid input or error is found -2: Fatal error is found (The system needs to be reset)
Description	Run Disconnect following description in topic 3.2.3.

4 Revision History

Revision	Date	Description
1.0	27-Mar-23	Initial version release