

TOE100G-IP on Alveo card reference design

Rev1.0 6-Jul-23

1	Introduction	2
2	TOE100DMATest (Hardware)	4
2.1	100G Ethernet Subsystem (100G BASE-SR)	5
2.2	TOE100G-IP	5
2.1	AxiDMACtrl512	6
2.1.1	MtMainCtrl	8
2.1.2	AxiMtPRd	11
2.1.3	AxiMtPWr	14
2.2	LAXI2Reg	20
2.2.1	SAXIReg	21
2.2.2	UserReg	23
3	The host software	27
3.1	Framework	28
3.1.1	Device interface	28
3.1.2	Shell	31
3.2	Application	37
3.2.1	Display parameters	39
3.2.2	Reset IP	39
3.2.3	Send data test	40
3.2.4	Receive data test	41
3.2.5	Full duplex test	43
3.2.6	Function list in application	45
4	Test Software on the target	51
4.1	“tcpdatatest” for half duplex test	51
4.2	“tcp_client_txx(_40G)” for full duplex test	53
5	Revision History	55

1 Introduction

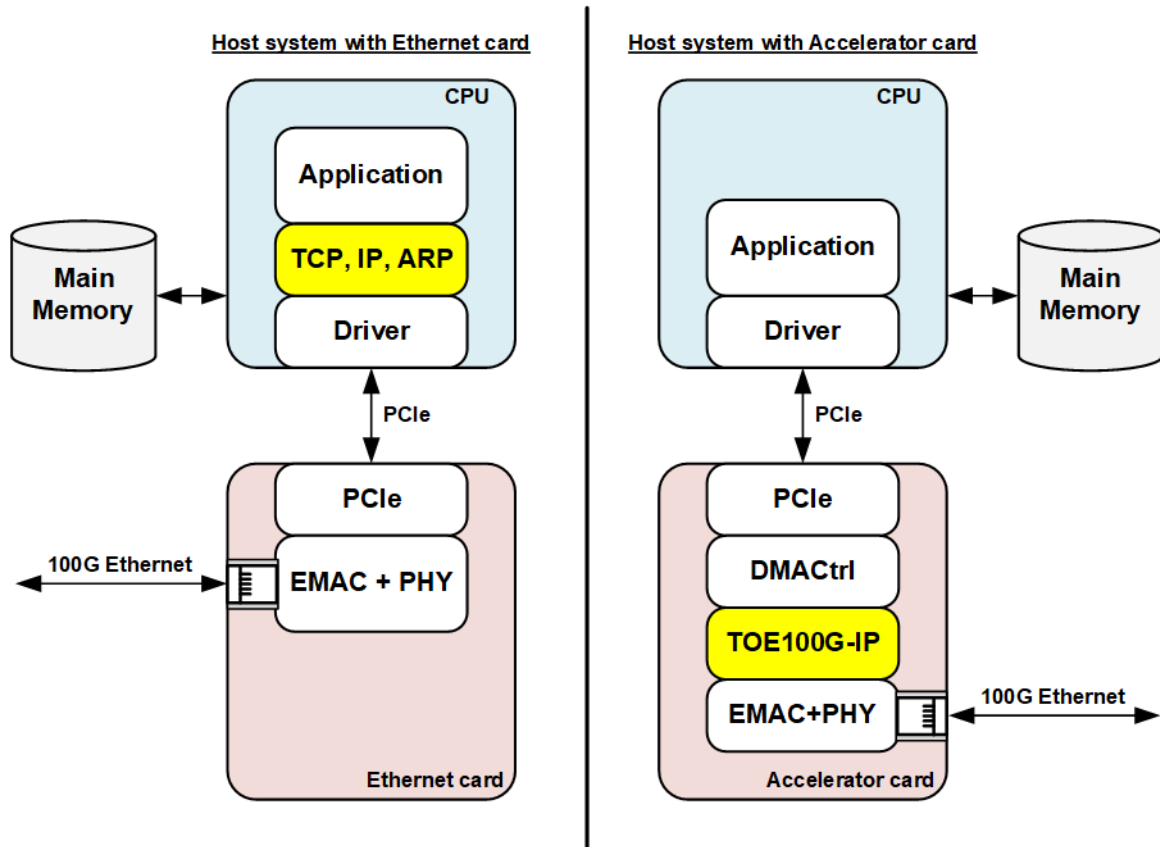


Figure 1-1 Ethernet card and Accelerator card comparison

The left side of Figure 1-1 shows the general solution when 100G Ethernet is required in the host PC. 100G Ethernet card is plugged-in to PCIe slot and then the user designs the test application on the software platform that has the library to handle TCP, IP, and ARP protocol. The device driver is provided by the 100G Ethernet card vendor to operate with the OS. Generally, the performance result on the test application when transferring the data via 100G Ethernet is limited by CPU task and the host system resource, so the maximum throughput of 100G Ethernet is not achieved.

The right side of Figure 1-1 is the Accelerator system that uses the Alveo Accelerator card to be the network interface card instead. TOE100G-IP by Design Gateway is integrated to be offload engine for TCP/IP protocol. TOE100G-IP offloads CPU to handle TCP, IP, and ARP protocol, so the TCP payload data is DMA transferred to Main memory, instead of Ethernet frame that is the output from EMAC. Also, the platform to transfer the data with the Main memory by using DMA engine via PCIe interface is specially developed by Xilinx. Therefore, the performance result when using the Accelerator card with TOE100G-IP is much better than using Ethernet card.

The TOE100G-IP on Alveo card reference design shows the complete solution on the hardware and the software to transfer TCP payload data on one TCP session with high-speed performance. The software application on the host system is developed by C++ languages, so it is easy for the user to integrate and modify the software to match with the system requirement.

Please see more details how to prepare the host system for running the Alveo accelerator card from the following site.

<https://www.xilinx.com/products/boards-and-kits/alveo.html>

Design Gateway also provides the development host system for Alveo card – Turnkey Accelerator System. Please check more details from our website.

<https://dgway.com/AcceleratorCards.html>

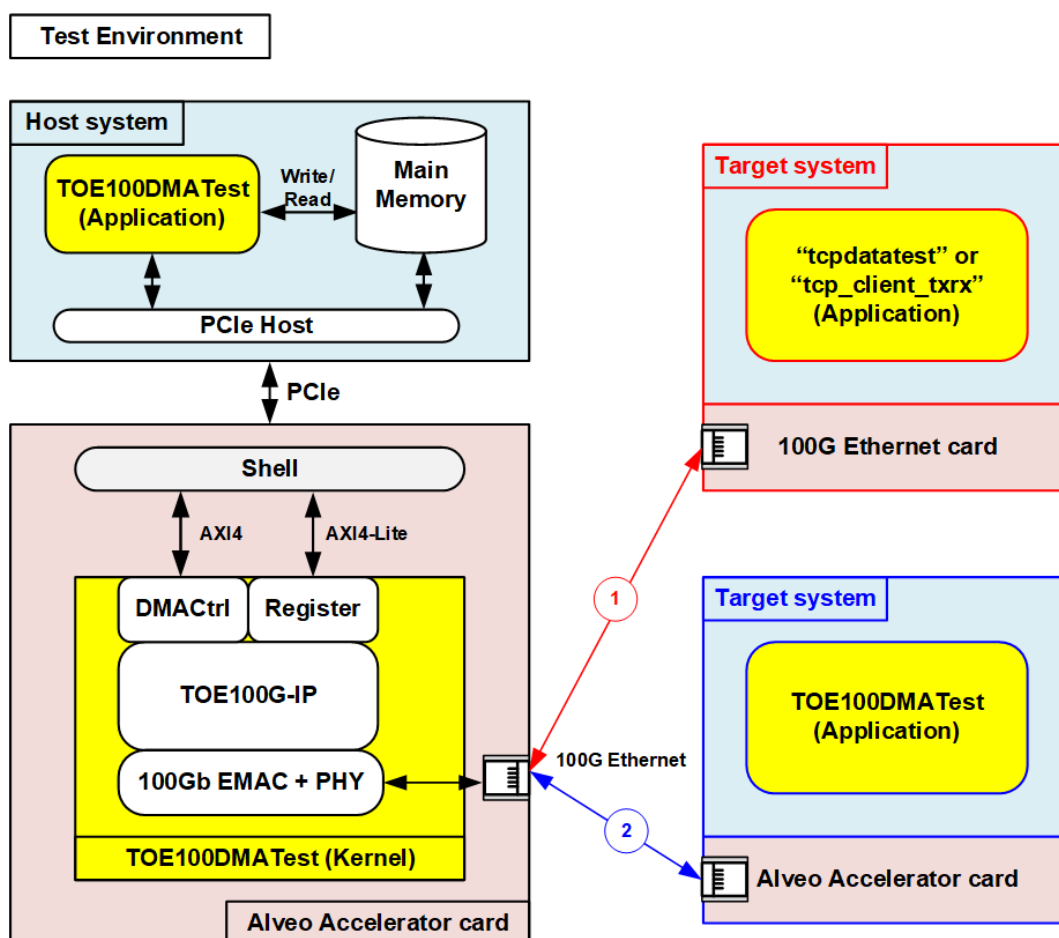


Figure 1-2 Test environment

The architecture of the TOE100G-IP demo on Alveo card can be separated into two systems - the host system which runs Ubuntu OS (Linux) and the hardware system which contains the TOE100DMA (FPGA logic). Two systems are connected through PCIe. The low layer on both the host system and the hardware system are handled and managed by Xilinx Runtime Library (XRT) and Vitis target platform. To transfer the TCP/IP payload data at 100Gb speed on the host system, the data is generated by the application with the memory allocation for transferring the data. The memory uses multiple buffering to maximize the TCP/IP transmission performance.

To run the demo, the target system can be either Test PC with Ethernet card or TOE100G-IP on another system. To use Test PC with Ethernet card, Design Gateway provides the test applications – “tcpdatatest” and “tcp_client_txrx” for the half-duplex transfer (send or receive data) and the full-duplex transfer (send and receive data at the same time by using one TCP session). The test applications are provided on both Windows 10 OS and Ubuntu 20.04 OS. However, using the test applications and the 100G Ethernet card show limited transfer performance. To achieve the maximum performance on 100G Ethernet, the target system should be TOE100G-IP on another system.

In the document, topic 2 shows the details of the hardware design on Alveo card. Topic 3 describes the software implementing on the Accelerator system. The last topic is the details of test application on the target system for half-duplex test and full-duplex test.

2 TOE100DMATest (Hardware)

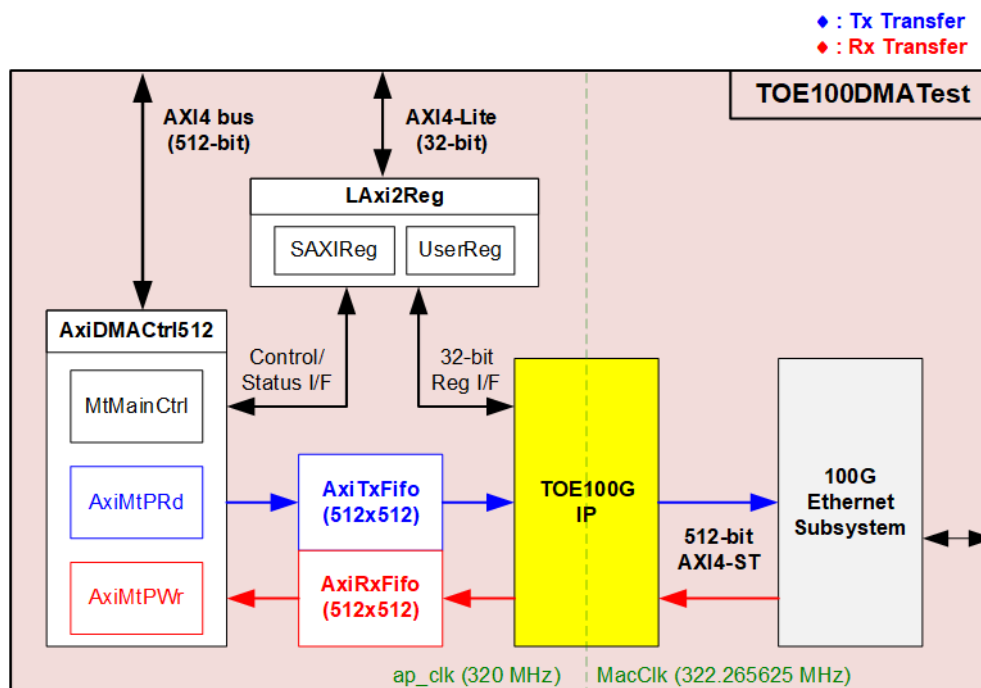


Figure 2-1 TOE100DMATest block diagram

The platform provides two interface types for the hardware kernel – AXI4 for transferring data with the Main memory and AXI4-Lite for register access which is generally applied to be control/status signals of the hardware kernel. AxiDMACtrl512 is the DMA engine for transferring data in two directions. The first one is to read the data from the Main memory and then forward to TOE100G-IP via FIFO (AxiTxFifo). After that, the TCP/IP packet is transmitted to the target system. The TCP payload data on the host memory is prepared by the test application. The second one is to receive the data from TOE100G-IP via FIFO (AxiRxFifo) and then write it to the Main memory. The test application on the host system reads the data from the Main memory with or without data verification.

AXI4-Lite bus, the platform interface, connects to LAXi2Reg module which is the adapter to convert AXI4-Lite bus to be Register interface for Control/Status signals. In TOE100DMATest kernel, it maps the register interface of AxiDMACtrl512 and TOE100G-IP to LAXi2Reg, so the test application can set the test parameters and monitor the test progress of the hardware.

TOE100G-IP connects with 100G Ethernet Subsystem via 512-bit AXI4-ST bus for connecting with the 100G Ethernet hardware connection. The Ethernet Subsystem uses MacClk domain which is equal to 322.266 MHz while the AXI interface of the platform uses ap_clk that is configured by the platform. In this demo, ap_clk is configured to be equal to 320 MHz for high-performance operation. CDC (Clock-crossing domain) is implemented inside TOE100G-IP. According to TOE100G-IP datasheet, clock frequency of user interface (ap_clk) must be more than or equal to 220 MHz. More details of each hardware module inside the TOE100DMATest are described as follows.

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

This module implements EMAC and PCS/PMA logic of 100G Ethernet. The physical interface on FPGA board can be applied by QSFP28 or 4xSFP28 for 100Gb BASE-SR standard. The user interface for connecting with EMAC is 512-bit AXI4-stream interface running at 322.265625 MHz. This IP core is generated by using Xilinx IP wizard. More details of the core are described in the following link.

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

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

Note: In this demo, 100G Ethernet Subsystem enables RS-FEC feature, so please confirm the network equipment of the test environment for running this demo that it can support RS-FEC.

2.2 TOE100G-IP

TOE100G-IP implements TCP/IP stack to be the offload engine for transferring TCP/IP packet with the network device. User interface has two signal groups, i.e., control signals and data signals. Register interface is applied to set control registers and monitor status signals while Data signals are accessed by using FIFO interface. The interface with 100G EMAC is 512-bit AXI4-ST interface. More details are described in datasheet.

https://dgway.com/products/IP/TOE100G-IP/dg_toe100gip_data_sheet_xilinx.pdf

2.1 AxiDMACtrlI512

The TOE100G-IP supports full-duplex transfer test, so it is possible that data on AXI4 bus is transferred in both directions at the same time for sending and receiving data with TOE100G-IP. The test application on the host system must allocate two buffers (Tx buffer and Rx buffer) for each transfer direction. To achieve the high performance, each buffer should be split to many areas to allow the hardware and the software to operate parallelly. This reference design, each buffer consists of four areas – quad buffering (Tx Buffer#0-#3 and Rx Buffer#0-#3), as shown in Figure 2-2.

Note: Typically, using double buffer should be enough for CPU and DMACtrl transferring the data with the Main memory at different area as parallel processing. However, four areas are applied to add more safe time gap when CPU or DMACtrl pauses data transmission for long time.

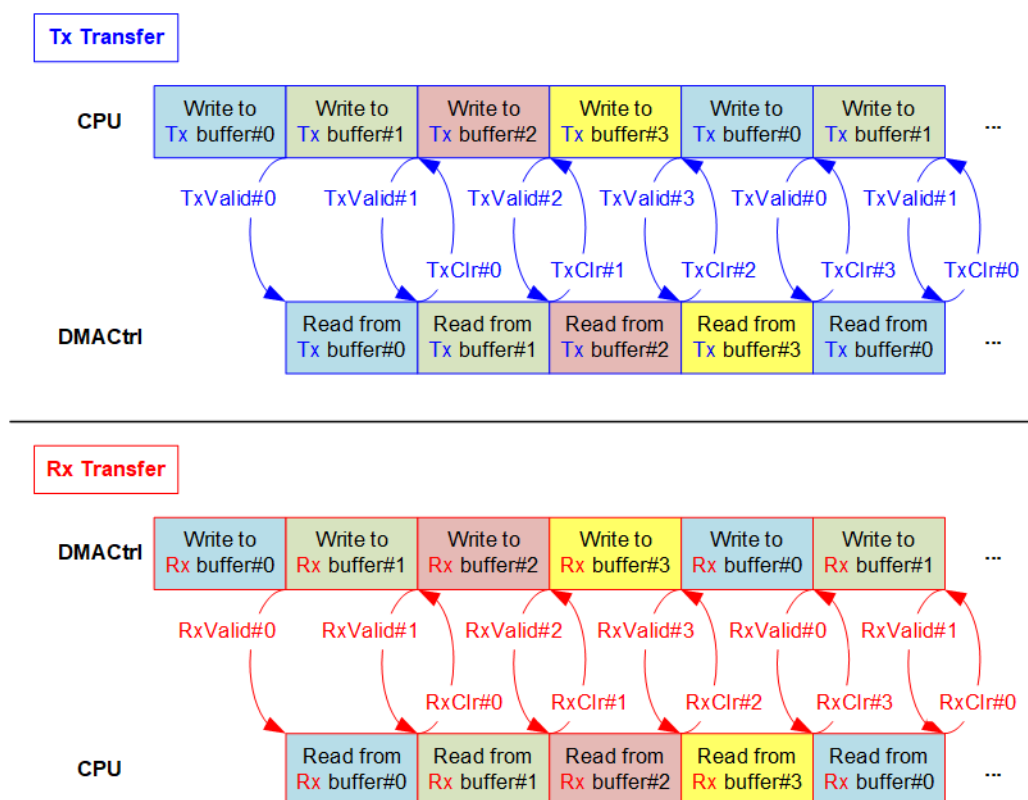


Figure 2-2 Quad buffering for DMA engine

The order to use each buffer area for both Tx and Rx Buffer is fixed to be #0 -> #1 -> #2 -> #3 -> #0.

For Tx transfer, the Main memory is written by the test application on the host system and read by the DMA engine (DMACtrl). The sequence of the operation is described as follows.

- 1) The CPU prepares the test data (dummy or incremental pattern) in the first area of the Main memory (Tx buffer#0). After filling the last data, the Valid status of this memory area (TxValid#0) is asserted.
- 2) If there is remaining data for transferring and the next memory area is free, the CPU starts writing the test data to the next area (Tx buffer#1). At the same time, DMACtrl detects the new memory area is valid and then it starts reading until the last data is read.
- 3) After finishing reading the last data, Clear status (TxClr#0) is asserted by DMACtrl to free the current memory area. If the next memory area is valid (TxValid#1 is asserted), DMACtrl starts the new operation. Repeat step 1) – 3) until total data is transferred.

On the other hand, the Main memory is written by DMACtrl and read by the test application for Rx transfer. The sequence of the operation is described as follows.

- 1) The DMACtrl prepares the test data in the first area of the Main memory (Rx buffer#0). After filling the last data, the Valid status of this memory area (RxValid#0) is asserted.
- 2) If there is remaining data for transferring and the next memory area is free, the DMACtrl starts writing the test data to the next area (Rx buffer#1). At the same time, CPU detects the new memory area is valid and then it starts reading until the last data is read.
- 3) After finishing reading the last data, Clear status (RxClr#0) is asserted by CPU to free the current memory area. If the next memory area is valid (RxValid#1 is asserted), CPU starts the new operation. Repeat step 1) – 3) until total data is transferred.

As Tx transfer and Rx transfer are operated individually, the AxiDMACtrl512 is designed by using three submodules for controlling Tx and Rx transfer separately, i.e., MtMainCtrl, AxiMtPRd, and AxiMtPWwr.

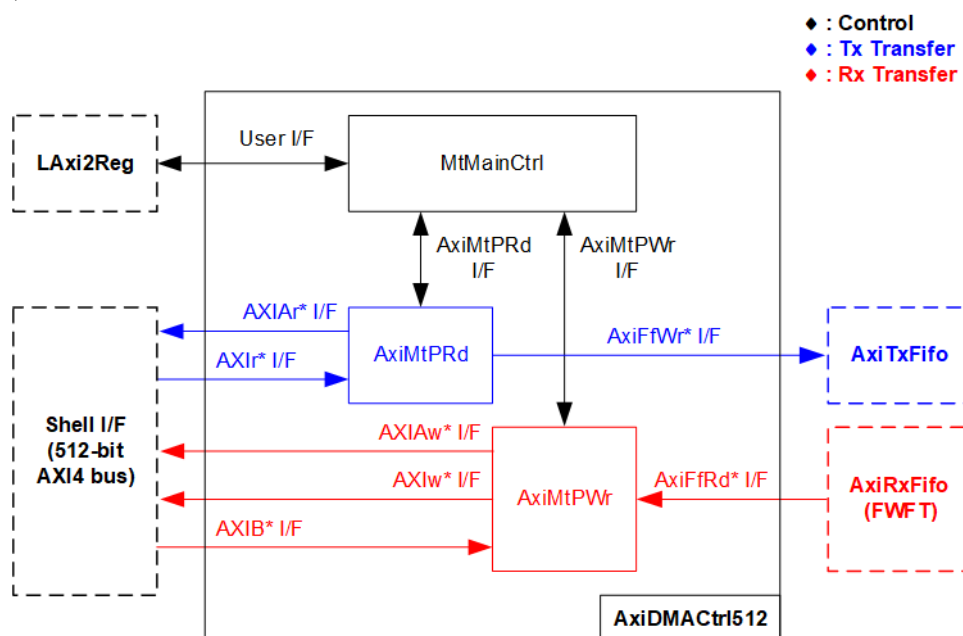


Figure 2-3 AxiDMACtrl512 Block Diagram

MtMainCtrl has the registers to store the test parameters, set by CPU. After decoding the test parameters, MtMainCtrl generates the request with the parameters for AxiMtPRd and AxiMtPWr to start Tx transfer and Rx transfer, respectively. AxiMtPRd generates the memory read request to the host system via AXI4 I/F to read the data from the Main memory (TxBuffer#0-#3) and transfer to AxiTxFifo. On the other hand, AxiMtPWr generates the memory write request to write data from AxiRxFifo to the Main memory (RxBuffer#0-#3) via AXI4 I/F. More details of each submodule are described as follows

2.1.1 MtMainCtrl

MtMainCtrl is designed to generate a command request to AxiMtPWr and AxiMtPRd for transfer the data of each buffer area. Therefore, when the total transfer size that is requested by the user is more than the buffer size area, multiple command requests are created by MtMainCtrl to AxiMtPWr/AxiMtPRd. It needs to have two individual submodules for generating command request to AxiMtPWr and AxiMtPRd. The operation to generate request to AxiMtPWr and AxiMtPRd are similar, so the same submodule, called AxiMtPCmd, is applied. This document shows the operation of AxiMtPCmd to control AxiMtPRd in Tx transfer and AxiMtPWr in Rx transfer by using timing diagram (Figure 2-4 and Figure 2-5).

The details of Figure 2-4 are described as follows.

- 1) UserTxStart is asserted by CPU to start reading the data from the Main memory. The first area to read is area#0 (TxBuffer#0). Thus, UserTxBufSel which shows the active buffer area is reset to 00b. The user input parameters – UserBufSize (buffer size of each area) and UserTxSize (total transfer size of this request) are loaded to the internal logic. UserTxBusy is asserted to '1' to show that the Tx request is accepted and the operation begins. The state enters to stChkSize.

Note: In Figure 2-4, total size value is equal to $N + N1$ to show that the transfer size of the last loop ($N1$) can be any value that is less than or equal to N (buffer size).
- 2) In stChkSize, the remaining transfer size (rTrnCnt) is read. If the read value is not equal to 0, the state continues to stChkBuf. Otherwise, the state returns to stIdle (step 9).
- 3) In stChkBuf, the buffer status (UserTxBufValid) of the active area is read. Each bit of UserTxBufValid is mapped to show the status of each area. When the active area is area#0, bit[0] is read. If UserTxBufValid is asserted to '1', the state enters to stGenCmd.
- 4) In stGenCmd, the parameters of AxiMtPRd I/F are prepared. AxiMtPRdAddr is equal to the start address of the active area of Tx buffer (TxBufAddr#0). Also, AxiMtPRdLen is equal to UserBufSize (N) for every run loop, except the last loop which is equal to the remaining value ($N1$). Then, AxiMtPRdReq is asserted to '1' to send the request to AxiMtPRd.
- 5) After that, AxiPRdBusy is asserted to '1' to confirm the request is accepted. The state enters to stWtEnd to wait until the operation of AxiPRd is done.
- 6) When AxiPRdBusy is de-asserted to '0', the state changes to the last state - stUpParam.
- 7) In stUpParam, the internal signals are updated. rTrnCnt decreases the value to show the remaining transfer length. UserTxBufSel is increased to show the next active area of Tx Buffer. UserTxBufClr of the active area (area#0) is asserted to '1' for one cycle to clear UserTxBufValid flag. Therefore, UserTxBufValid of the active area is de-asserted in the next clock cycle. Next, the state returns to stChkSize.
- 8) The next state is determined by rTrnCnt Value.
 - a. If $rTrnCnt \neq 0$, repeat step 2) – 7) to read the data from the next area of Tx buffer. When the next buffer is TxBuffer#1, use bit1 of UserTxBufValid and UserTxBufClr to operate.
 - b. If $rTrnCnt = 0$, the state returns to stIdle. After that, UserTxBusy is de-asserted to '0'.

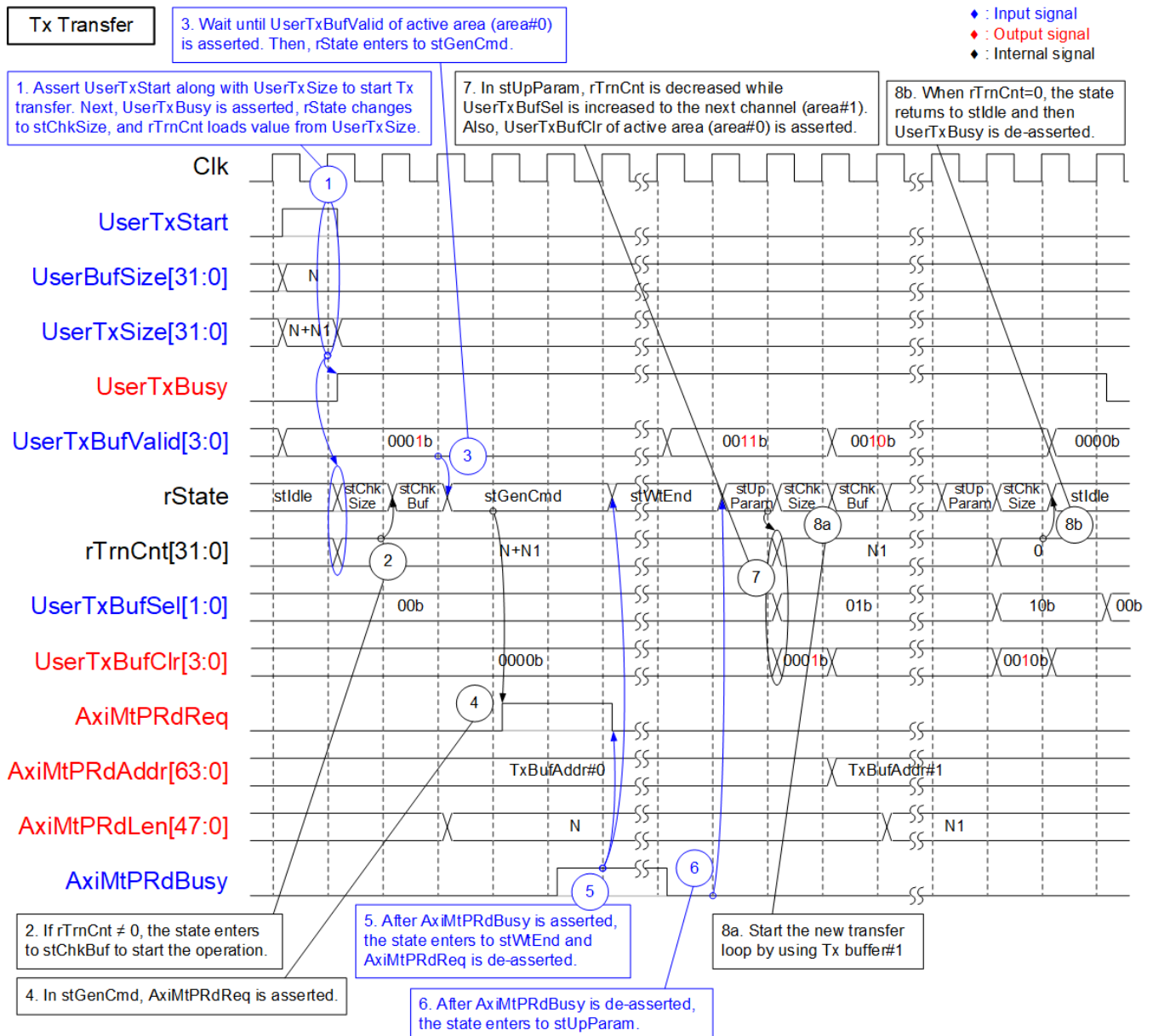


Figure 2-4 Tx transfer of MtMainCtrl timing diagram

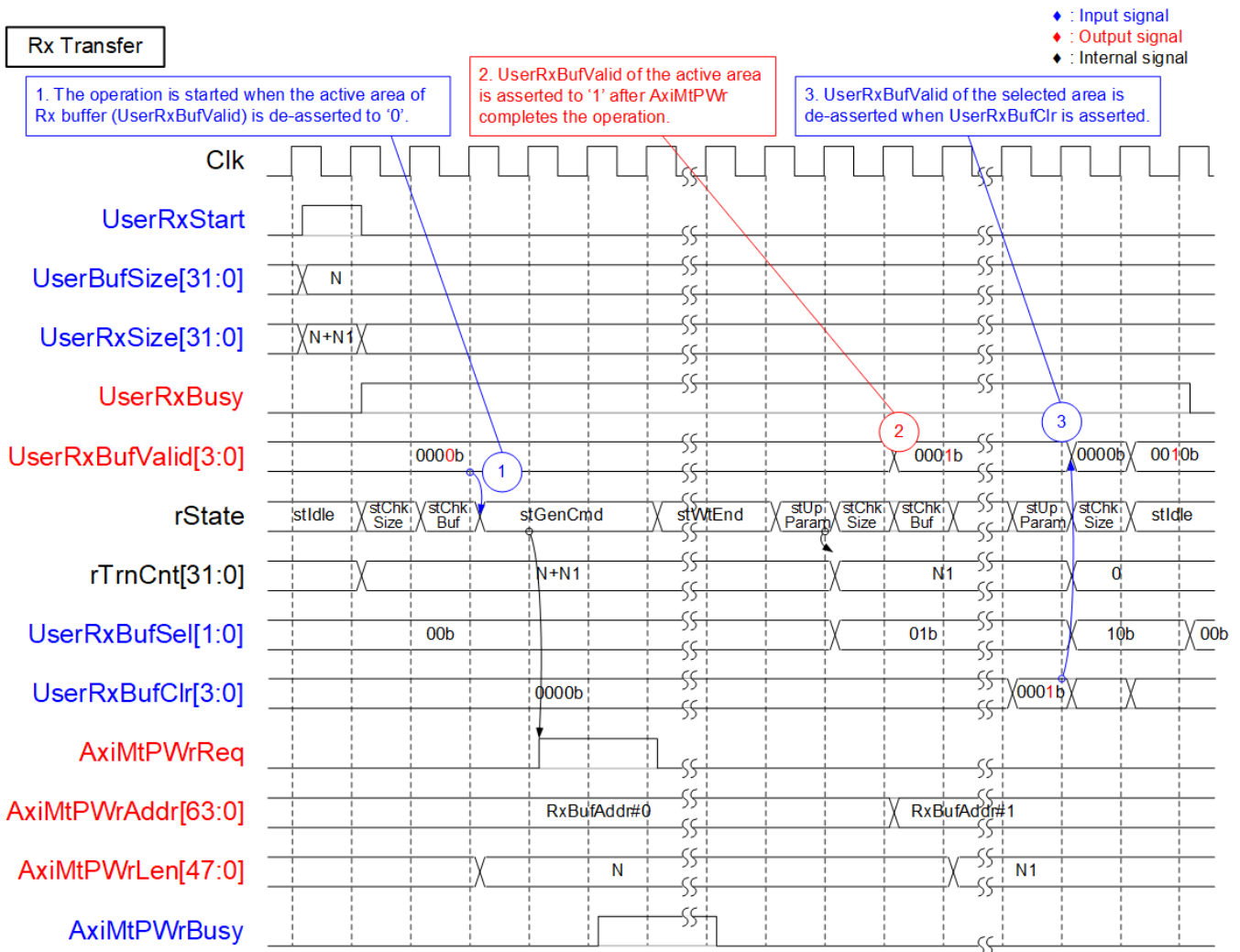


Figure 2-5 Rx transfer of MtMainCtrl timing diagram

Figure 2-5 shows the details for Rx transfer operation which the request is generated to AxiMtPWr and Rx buffer is applied. The work flow of Rx transfer is almost similar to Tx transfer, but it uses Rx parameter and AxiMtPWr interface. The description in Figure 2-5 shows only the different point between Rx transfer and Tx transfer. The transfer direction of Rx buffer is inversed from Tx buffer, so UserRxBufValid is asserted by MtMainCtrl and UserRxBufClr is the input from CPU.

- 1) In stChkBuf, UserRxBufValid of the active area must be de-asserted to '0' (no data available) before asserting the request to AxiMtPWr. After that, the data is written to the Main memory.
- 2) After AxiMtPWr finishes writing the data in each loop, UserRxBufValid of the active area is asserted to '1'.
- 3) When CPU finishes reading the data from the Main memory, UserRxBufClr is asserted. After that, the buffer status is empty.

2.1.2 AxiMtPRd

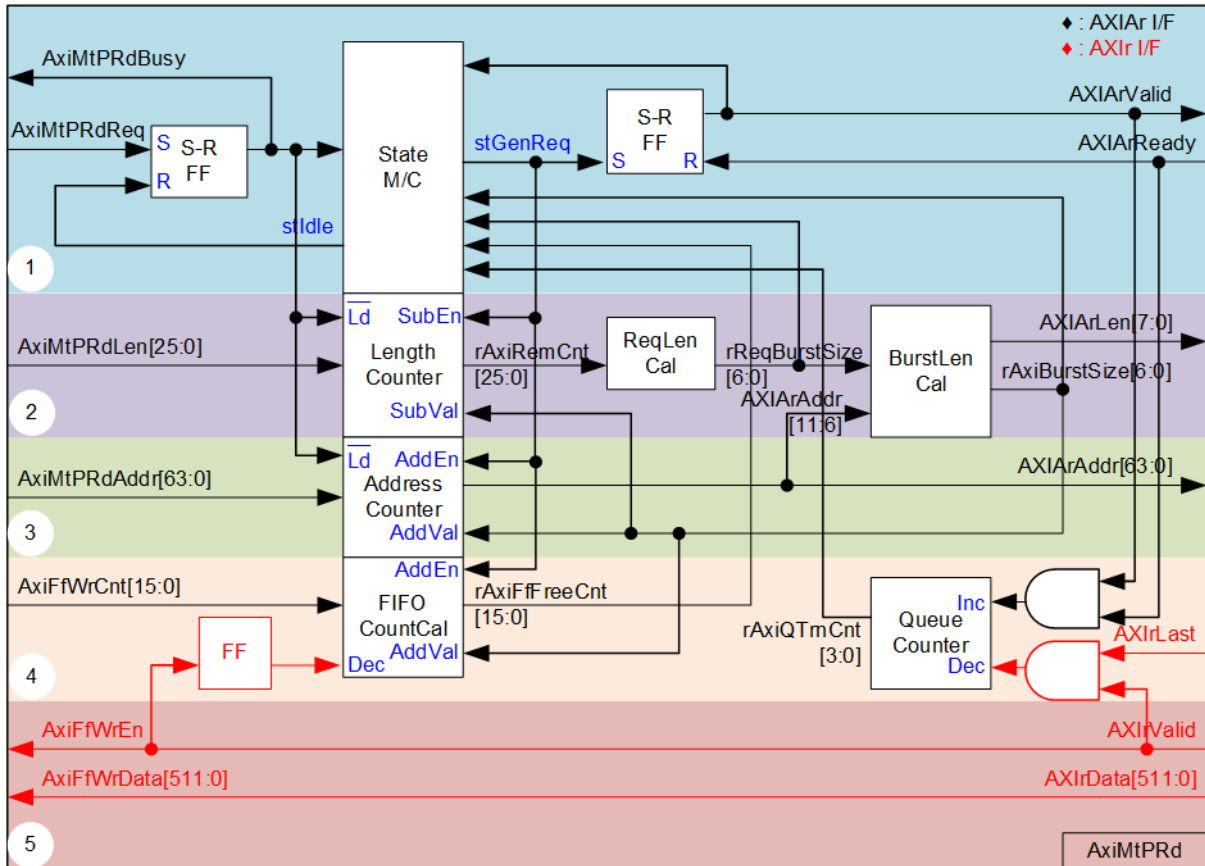


Figure 2-6 AxiMtPRd Block diagram

According to AXI4 standard, AXIAr I/F which is applied to send the read command request and AXIr I/F which is applied to transfer the data stream can be operated parallelly. Therefore, the logic inside AxiMtPRd is designed to send the new read command request via AXIAr I/F without waiting the data returned via AXIr I/F to achieve the best performance.

The operation is started when the user asserts AxiMtPRdReq along with AxiMtPRdLen (Total transfer length in 64-byte unit) and AxiMtPRdAddr (Start address of the Main memory in byte unit). After that, the read command request (AXIArValid) is generated by the State machine (Block no.1). Block no.2 is the logic to set the transfer size (AXIArLen) of each command request which can be equal to three values – 1, 8, or 64. Length Counter loads the total length from user (AxiMtPRdLen) before starting the operation. It calculates the remaining transfer size (rAxiRemCnt) after generating each request to AXIAr I/F. The remaining transfer size is fed to ReqLenCal to find the maximum transfer size of each request (rReqBurstSize). However, it needs to check the current address (AXIArAddr[11:6]) to confirm that this transfer does not cross the address boundary, designed by BurstLenCal block. The actual transfer size (AXIArLen) may be less than rReqBurstSize value if the current address is not aligned to 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 is the flow control logic to pause the new request that is generated by State machine. Two factors must be calculated before sending the new request. First is the actual free space size that is available in AxiTxFIFO (rAxiFfFreeCnt), calculated by FIFOCountCal. The current value of FIFO data counter (AxiFfWrCnt) is read and then added by the amount of data that is requested but does not transfer to be the usage size. The free space size is computed by using NOT logic to the used size. The new request is generated when the free space size is enough for storing the data of the new request. The second factor is the number of command request that does not receive the data (rAxiQTrnCnt), calculated by Queue Counter. The counter is up-counted when the new command request is asserted and down-counted when the last data of each request is received. AxiMtPRd uses 4-bit Queue counter, so State machine can create up to 15 commands (rAxiQTrnCnt=1111b) without waiting the new data is transferred on AxiRData. The last block, Block no.5, shows the data path that is directly mapped from AxiR I/F to AxiFIFO I/F.

Figure 2-7 shows the timing diagram of AxiMtPRd logic when the user command request sets total transfer size to 65 and the start address on AxiPRdAddr (A0) is aligned to 64-byte unit. Two read command requests are generated by AxiMtPRd. First request is 64-beat transfer while the second request is 1-beat transfer. The details are described as follows.

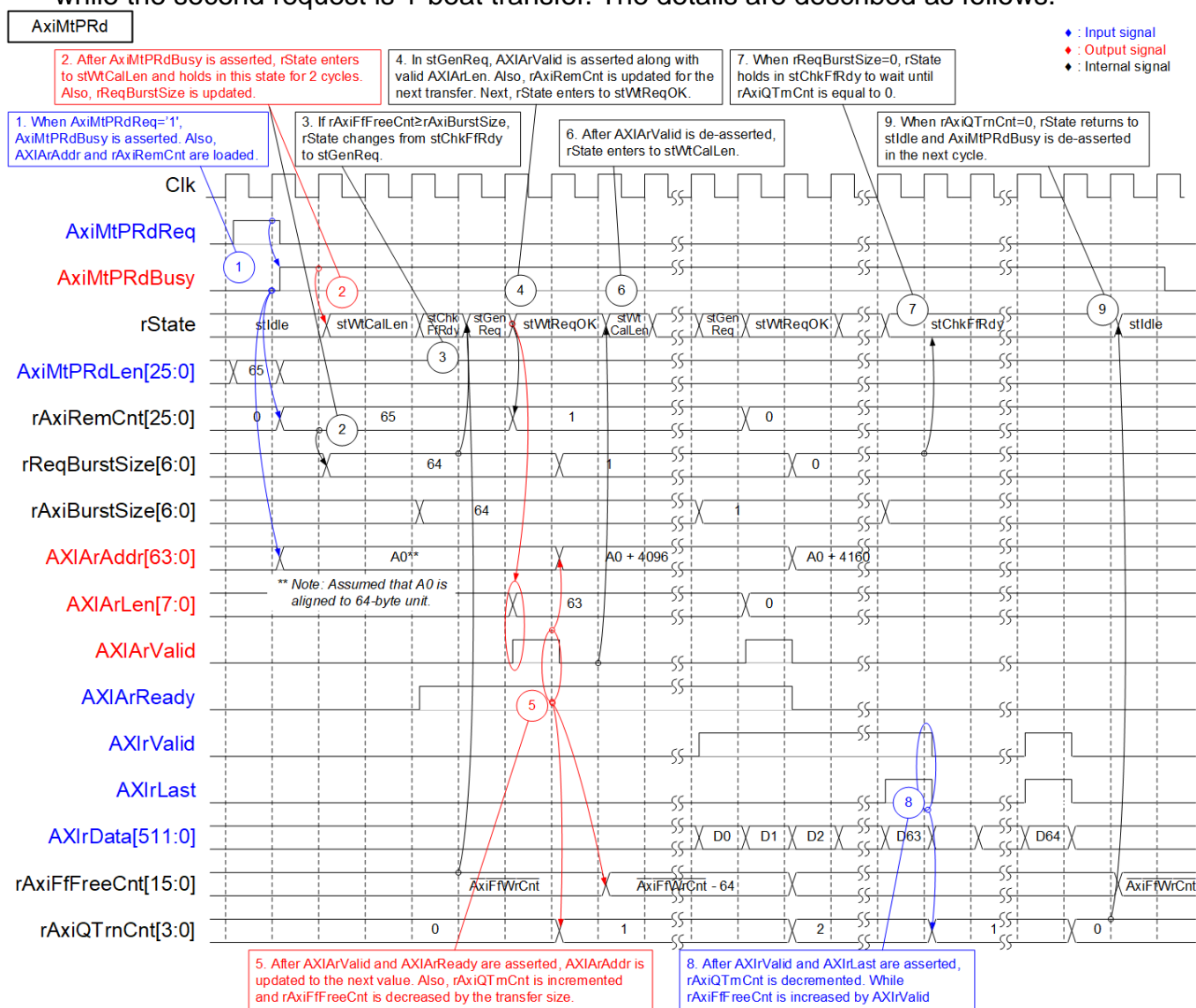


Figure 2-7 AxiMtPRd Timing diagram

- 1) The new command request (AxiMtPRdReq) is asserted to '1' along with the valid AxiMtPRdAddr (the start address of Main memory) and AxiMtPRdLen (total transfer size in 64-byte unit). AxiMtPRd asserts AxiMtPRdBusy to '1' to accept the request. Meanwhile, AXIArAddr loads the initial value from AxiMtPRdAddr and rAxiRemCnt loads the initial value from AxiMtPRdLen.
- 2) After AxiMtPRdBusy is asserted, rState enters to stWtCallLen to start calculating the transfer size of this command request, sent to AXIAr I/F. If rAxiRemCnt is more than or equal to 64, rReqBurstSize (the maximum request size) is set to 64. Otherwise, it is set by rAxiRemCnt. rReqBurstSize and the lower bit of AxiArAddr are read to calculate rAxiBurstSize. rAxiBurstSize may be less than rReqBurstSize if the lower bit of AxiArAddr is not aligned to rReqBurstSize. The state holds in stWtCallLen for two clock cycles to wait until rAxiBurstSize is valid before entering to stChkFfRdy.
- 3) In stChkFfRdy, it holds in this state to wait until the FIFO has enough free space for this transfer ($rAxiFfFreeCnt \geq rAxiBurstSize$) when there is remaining command request to generate ($rAxiRemCnt \neq 0$). After that, it enters to stGenReq.
Note: Step 7 shows the example when all command request is generated.
- 4) The new command request is generated when rState enters to stGenReq which is one-cycle state. In the next clock, rState enters to stWtReqOK. AXIArValid is asserted and AXIArLen loads the value from the calculation unit. AXIAr I/F output signals hold the value until the request is accepted by asserting AXIArReady to '1'.
- 5) When the command request is accepted ($AXIArValid='1'$ and $AXIArReady='1'$), AXIArAddr is updated to the next value and rAxiQTrnCnt is incremented. While rAxiFfFreeCnt is decreased by the current transfer size (64).
Note: rAxiFfFreeCnt is decreased when the request is sent to compensate the amount of the data that does not received from the latest request.
- 6) After AXIArValid is de-asserted to '0', rState returns to stWtCallLen to wait until the next transfer size (rAxiBurstSize) is valid. Step 2) – 6) are repeated to generate the second request to AXIAr I/F.
- 7) In stChkFfRdy, if there is no more command request to generate ($rReqBurstCnt=0$), rState holds the value to wait until all data are transferred completely ($rAxiQTrnCnt=0$).
- 8) The data interface is run independently. The data transfer can be started any time. When the last data of each request is received ($AXIrLast='1'$ and $AXIrValid='1'$), rAxiQTrnCnt is decremented. While rAxiFfFreeCnt is incremented every cycle that AXIrValid is asserted. Finally, when the last data is received, rAxiQTrnCnt is equal to 0 and rAxiFfFreeCnt (the free size) is equal to the NOT value of AxiFfWrCnt (the used size).
- 9) After the operation is done, rState returns to stIdle and then AxiRdBusy is de-asserted to '0'. After that, MtMainCtrl can send the new command to AxiMtPRd.

2.1.3 AxiMtPWrr

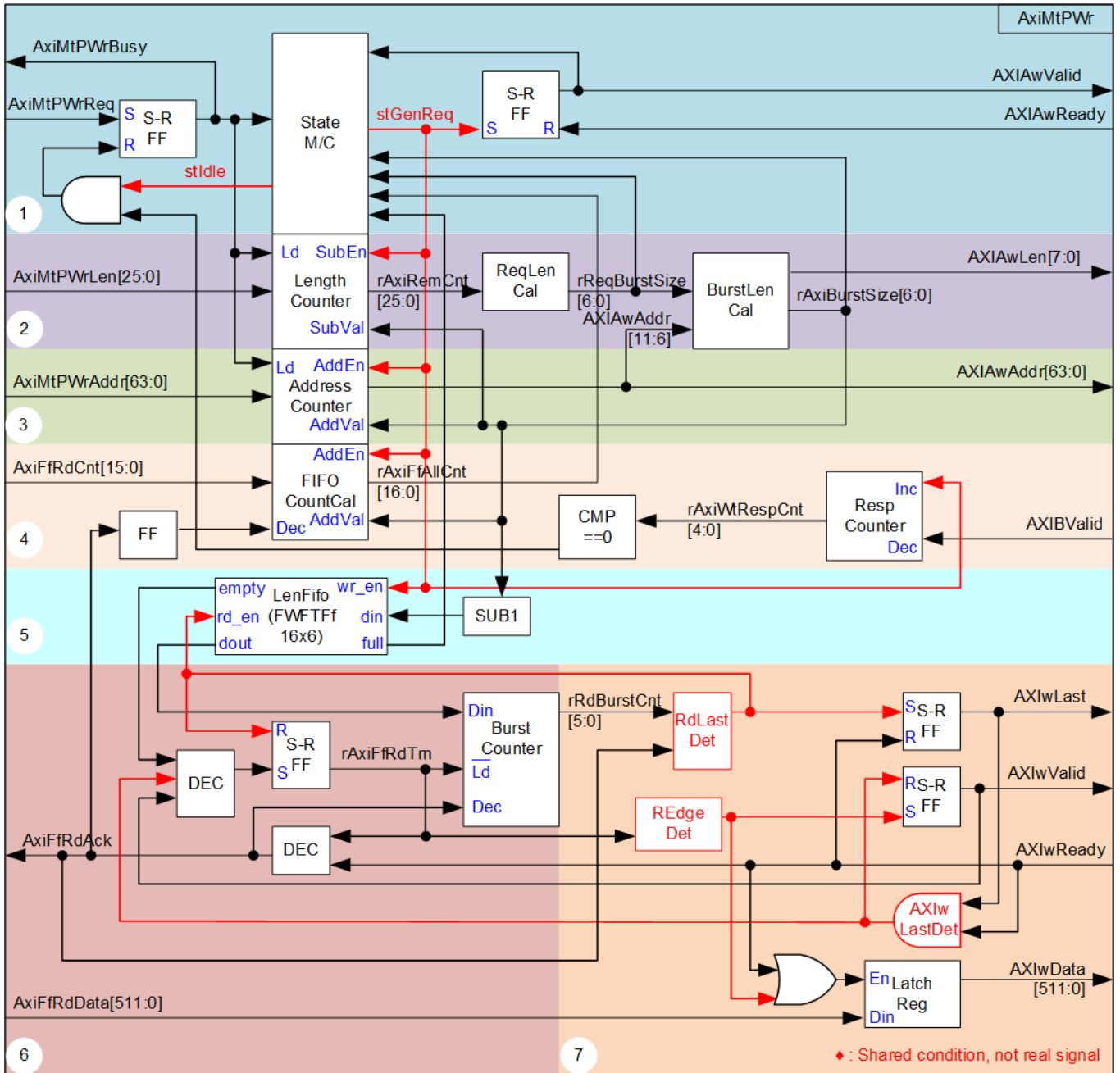


Figure 2-8 AxiMtPWrr Block diagram

Similar to AxiMtPRd module, the write command request via AXIAw I/F can be generated without starting data transferring via AXIw I/F. Therefore, the logics for controlling the AXIAw I/F and AXIw I/F are designed individually. As shown in Figure 2-8, the logic inside AxiMtPWrr can be divided to three parts. First is the logic for interface with AXIAw I/F which is shown in Block no.1 – no.4. Second is Block no.5 which is the small FWFT FIFO that stores the transfer length which is requested to AXIAw I/F for controlling data transfer in the AXIw I/F. Finally, Block no.6 and no.7 are the logic for interface with AXIw I/F.

Block no.1 is the State machine that is the core engine to generate the write command request (AXIAwValid) to AXIAw I/F. The operation of the State machine is almost similar to the State machine inside AxiMtPRd. The busy flag (AxiMtPWrBusy) is asserted after the user request (AxiMtPWrReq) is asserted to '1'. The operation is done and AxiMtPWrBusy is de-asserted when all write responses are received via AXIB I/F. Block no.2 is the logic to calculate the transfer length (AXIAwLen) of each command request. This block uses the same logic as Block no.2 inside AxiMtPRd module. Three transfer sizes are supported – 1, 8, or 64, depending on the remaining transfer size (rAxiRemCnt) and the lower bit of the current address (AxiAwAddr[11:6]). Block no.3 - the address counter also uses the same logic as AxiMtPRd module. Block no.4 is the flow control logic to determine the amount of data that is stored in AxiRxFIFO. The actual remaining data size (rAxiFfAllCnt) is calculated by reading the current data count of AxiRxFIFO (AxiFfRdCnt) and then subtracted by the amount of data that is requested but does not transfer. The new command request can be generated when rAxiFfAllCnt is more than or equal to rAxiBurstSize. While AxiMtPRd uses Queue Counter, AxiMtPWr uses Resp Counter to check if the operation is done. The Resp Counter is incremented when the Write request is asserted and then decremented when the write response is received (AXIBValid='1'). If rAxiWtRespCnt = 0, the data of all command request is transferred completely.

As shown in Block no.5, small FWFT FIFO is integrated to store the transfer length of each Write request to AXIAw I/F. The data interface inside Block no.6 reads this value to control the transfer size of each data transfer loop. FIFO depth is 16, so the State machine can generate up to 16 write command requests (FIFO depth) without starting transferring data to AXIw I/F.

Block no.6 is the logic which loads the transfer length of each request from LenFifo and then reads the data from AxiRxFIFO for transferring to AXIw I/F. AxiRxFIFO is FWFT type, so the read enable is called AxiFfRdAck (read acknowledge). The core signal of this block is rAxiFfRdTrn which is asserted while transferring the data. The operation begins by asserting rAxiFfRdTrn to '1' when empty flag of LenFIFO is de-asserted and the data interface returns to Idle (AXIwValid='0' or the last data is sent). The operation is done by de-asserting rAxiFfRdTrn to '0' when the last data is read from AxiRxFIFO (RdLastDet is found). Burst Counter is designed to count the transfer size of each request. It loads the initial value from LenFIFO and then decreases the value when each data is read from AxiRxFIFO (AxiFfRdAck='1'). The last data is detected by monitoring rRdBurstCnt=0. When AXIw I/F is not ready to receive the data (AXIwReady='0'), AxiFfRdAck is de-asserted to pause reading the next data from AxiRxFIFO.

Lastly, Block no.7 is the output register to transfer the data to AXIw I/F. AXIwValid is always asserted to '1' from the first data to the final data of each request size. Therefore, it is asserted when detecting rising edge of rAxiFfRdTrn (the data of new request is started). Also, it is de-asserted to '0' after transferring the final data completely. AXIwLast is asserted to '1' when the last data is read from AxiRxFIFO. To synchronous with AXIwValid, the read data from AxiRxFIFO (AxiFfRdData) must store to Latch register before forwarding to AXIwData.

Figure 2-9 shows timing diagram of command interface (AXIAw I/F) which is controlled by the State machine. It shows the example when user sets transfer size to 65 and the start address (AxiMtPWrAddr) is aligned to 64-byte unit. Therefore, two write command requests are created – 64-beat transfer and 1-beat transfer, respectively.

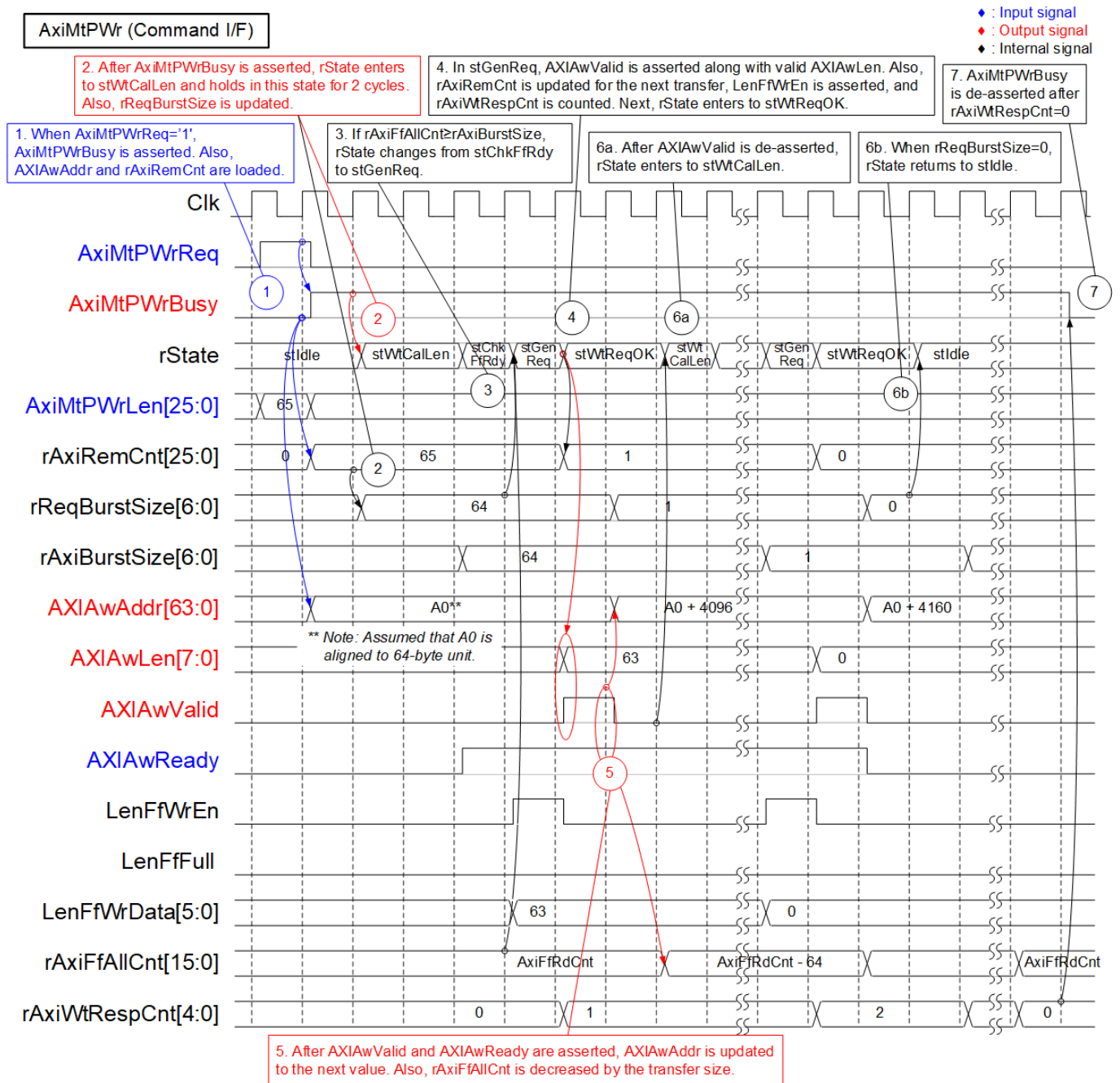


Figure 2-9 Command I/F of AxiMtPWr Timing diagram

- 1) The new command request (AxiMtPWrReq) is asserted to '1' along with the valid AxiMtPWrAddr (the start address of Main memory) and AxiMtPWrLen (total transfer size in 64-byte unit). AxiMtPWr asserts AxiMtPWrBusy to '1' to accept the request. Meanwhile, AXIAwAddr loads the initial value from AxiMtPWrAddr and rAxiRemCnt loads the initial value from AxiMtPWrLen.
- 2) After AxiMtPWrBusy is asserted, rState enters to stWtCallLen to start calculating the transfer size of this command request to AXIAw I/F. If rAxiRemCnt is more than or equal to 64, rReqBurstSize (the maximum request size) is set to 64. Otherwise, it is set by rAxiRemCnt. rReqBurstSize and the lower bit of AxiAwAddr are read to calculate rAxiBurstSize. rAxiBurstSize may be less than rReqBurstSize if the lower bit of AxiArAddr is not aligned to rReqBurstSize. The state holds in stWtCallLen for two clock cycles to wait until rAxiBurstSize is valid before entering to stChkFfRdy.
- 3) In stChkFfRdy, it holds in this state to wait until the FIFO has enough data for this transfer ($rAxiFfAllCnt \geq rAxiBurstSize$). After that, it enters to stGenReq.
- 4) The new command request is generated when rState enters to stGenReq which is one-cycle state. In the next clock, rState enters to stWtReqOK. AXIAwValid is asserted and AXIAwLen loads the value from the calculation unit. AXIAw I/F output signals hold the value until the request is accepted by asserting AXIAwReady to '1'. While $rState = stGenReq$, the transfer length ($rAxiBurstSize - 1$) is written to LenFifo by asserting LenFfWrEn to '1'. Also, rAxiWtRespCnt is incremented.
- 5) When the command request is accepted ($AXIAwValid = '1'$ and $AXIAwReady = '1'$), AXIAwAddr is updated to the next value and rAxiFfAllCnt is decreased by the current transfer size (64).
Note: rAxiFfAllCnt is decreased when the request is sent to compensate the amount of the data that does not received from the latest request. After each data is received from AxiRxFIFO (AxiFfRdAck='1'), rAxiFfAllCnt is incremented to reduce the amount of compensated data.
- 6) After AXIAwValid is de-asserted to '0', rState reads the next request size (rReqBurstSize).
 - a. If $rReqBurstSize \neq 0$, it returns to stWtCallLen and step 2) – 6) are repeated for operating the next request.
 - b. If $rReqBurstSize = 0$, rState returns to stIdle.
- 7) After all data are transferred completely and all responses are received, rAxiWtRespCnt is equal to 0 and then rAxiWrBusy is de-asserted to '0'. After that, MtMainCtrl can send the new command to AxiMtPWr.

Figure 2-10 shows timing diagram of data interface (AXIw I/F) which is designed to transfer the data from AxiRxFIFO which is FWFT type to AXIw I/F. The example shows two data transfers, i.e., 64-beat transfer and 1-beat transfer, matched to Figure 2-9.

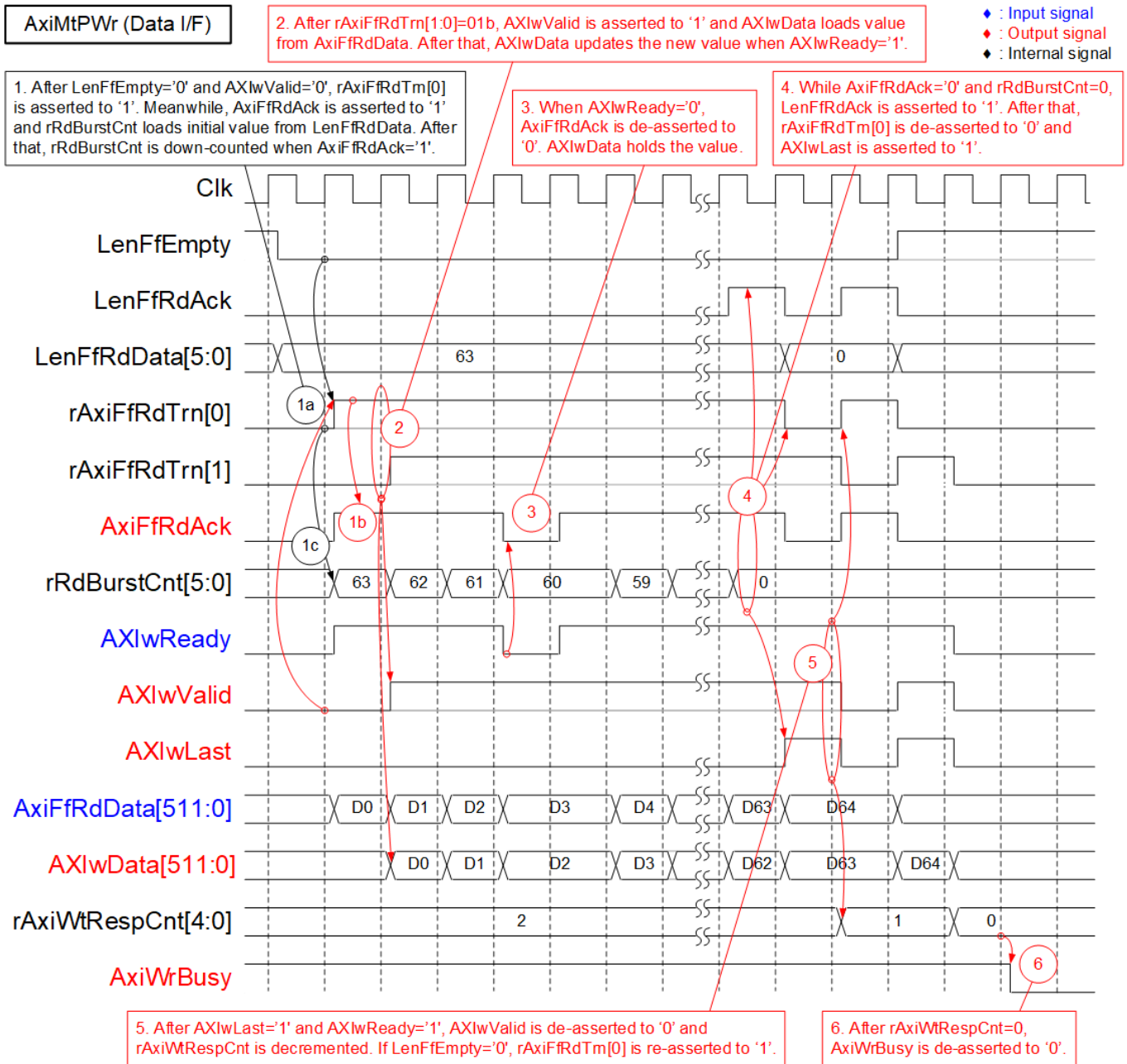


Figure 2-10 Command I/F of AxiMtPWr Timing diagram

- 1) After the new command request is generated to AXIw I/F, the empty flag of LenFifo (LenFfEmpty) is de-asserted to '0'. The data interface starts the operation by asserting rAxiFfRdTrn[0] to '1'. Before asserting rAxiFfRdTrn[0], it needs to match one of two conditions. The data interface is Idle (AXIwValid='0', shown in this step) or the last data is completely transferred (AXIwValid='1' and AXIwLast='1', shown in step 5). At the same time as rAxiFfRdTrn[0] asserted, AxiFfRdAck is also asserted to '1' to read the next data from AxiRxFIFO. While there is no data transferring, rRdBurstCnt always loads the initial value from LenFfRdData. After that, rAxiFfRdTrn[0] is always asserted to '1' until reading the last data of this transfer (D63) from AxiRxFIFO. While rRdBurstCnt is down-counted when AxiFfRdAck='1' to show the amount of remaining data in this transfer.
- 2) Next, rAxiRdTrn[1] which is rAxiRdTrn[0] with one-clock latency is asserted to '1'. By detecting the rising edge of rAxiRdTrn[0], it mentions that the first data from AxiRxFIFO is ready. Therefore, AXIwValid is asserted to '1' and AXIwData loads the value from AxiFfRdData. After that, AXIwValid is always asserted to '1' until the last data is transferred. AXIwData loads the remaining data when the new data is read from AxiRxFIFO by checking AXIwReady='1' (AXIwReady='1' can be represented to AxiFfRdAck='1').
- 3) When AXIw I/F is not ready to receive the data by de-asserting AXIwReady to '0', the data interface must pause the transmission by de-asserting AxiFfRdAck to '0'. AXIwData and other AXIw I/F signals hold the same value until AXIwReady is re-asserted to '1'.
- 4) As the last data is read from AxiRxFIFO (AxiFfRdAck='1' and rRdBurstCnt=0), LenFfRdAck is asserted to '1' to flush the current data and read the next data from LenFifo. In the next clock, rAxiFfRdTrn[0] is de-asserted to '0' and the last data is transferred to AXIw I/F (AXIwLast='1' and AXIwData=D63).
- 5) After the last data is accepted by AXIw I/F (AXIwLast='1' and AXIwReady='1'), AXIwValid is de-asserted to '0' and rAxiWtRespCnt is decremented. Also, if LenFifo has more data (LenFfEmpty='0'), step 1 – step 5 are repeated to start the next data transfer which is 1-beat transfer.
- 6) After all data is completely transferred, rAxiWtRespCnt is equal to zero. Next, AxiWrBusy is de-asserted to '0' to show user that the operation is done.

2.2 LAXi2Reg

AXI4-Lite is the interface of the hardware kernel for accessing the hardware registers. CPU uses this interface to set the parameters to the hardware and also monitor the hardware status while operating. 32-bit data bus size is applied. LAXi2Reg is run on application clock domain that is configured by the tool.

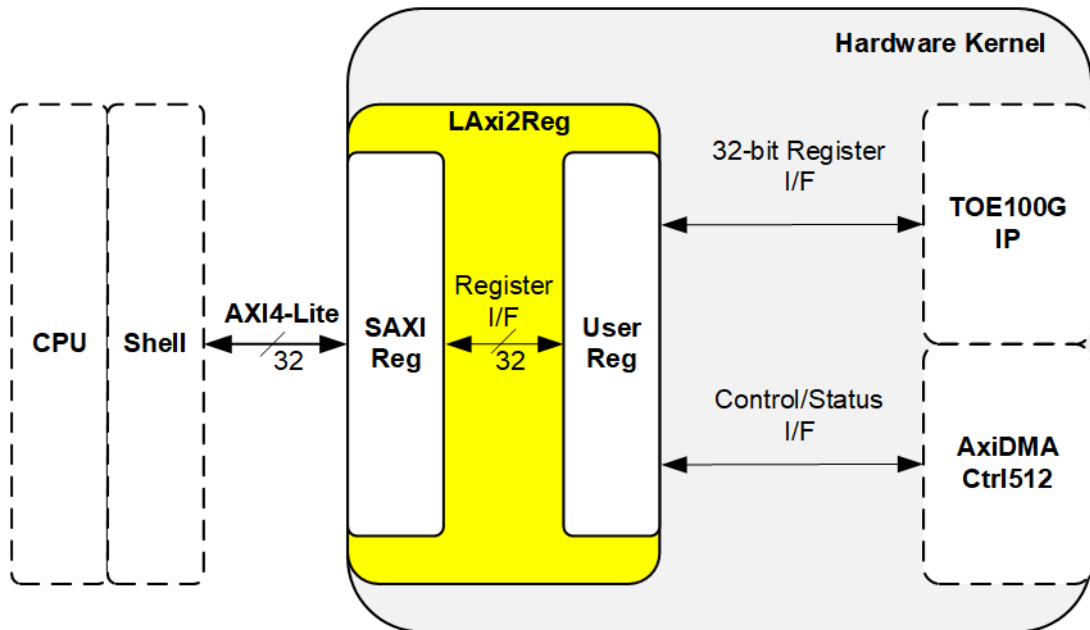


Figure 2-11 LAXi2Reg interface

LAXi2Reg consists of SAXIReg and UserReg. SAXIReg converts the AXI4-Lite signals to be the simple register interface which has 32-bit data bus size (similar to AXI4-Lite data bus size). UserReg includes the register file of the parameters and the status of the submodules. More details of SAXIReg and UserReg are described as follows.

2.2.1 SAXIReg

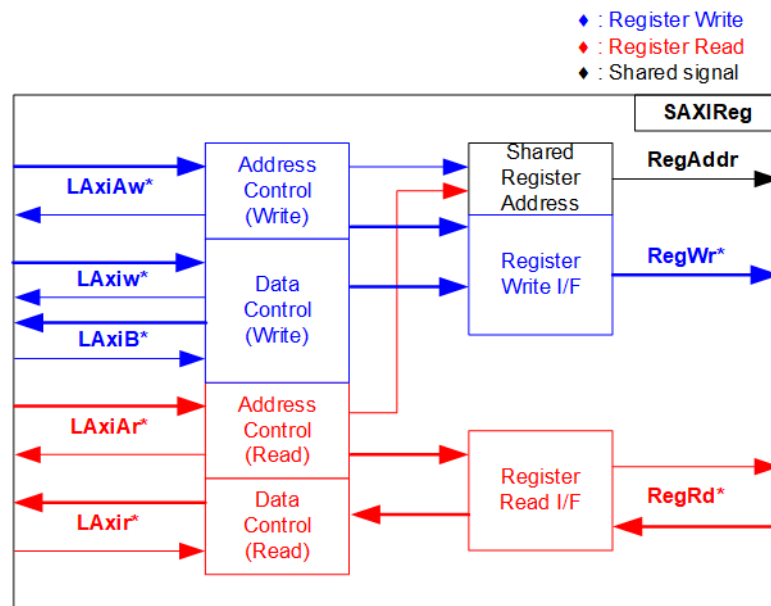


Figure 2-12 SAXIReg Interface

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

https://forums.xilinx.com/xlnx/attachments/xlnx/NewUser/34911/1/designing_a_custom_axi_slave_rev1.pdf

According to AXI4-Lite standard, the write channel and the read channel are operated independently. Also, the control and data interface of each channel are run separately. The logic inside SAXIReg to interface with AXI4-Lite bus is split into four groups, i.e., Write control logic, Write data logic, Read control logic, and Read data logic as shown in the left side of Figure 2-12. Write control I/F and Write data I/F of AXI4-Lite bus are latched and transferred to be Write register interface. Similarly, Read control I/F of AXI4-Lite bus are latched and transferred to be Read register interface. While the returned data from Register Read I/F is transferred to AXI4-Lite bus. In register interface, RegAddr is shared signal for write and read access. Therefore, it loads the address from LAXiAw for write access or LAXiAr for read access.

The simple register interface is compatible with single-port RAM interface for write transaction. The read transaction of the register interface is slightly modified from RAM interface by adding RdReq and RdValid signals for controlling read latency time. The address of register interface is shared for write and read transaction, so user cannot write and read the register at the same time. The timing diagram of the register interface is shown in Figure 2-13.

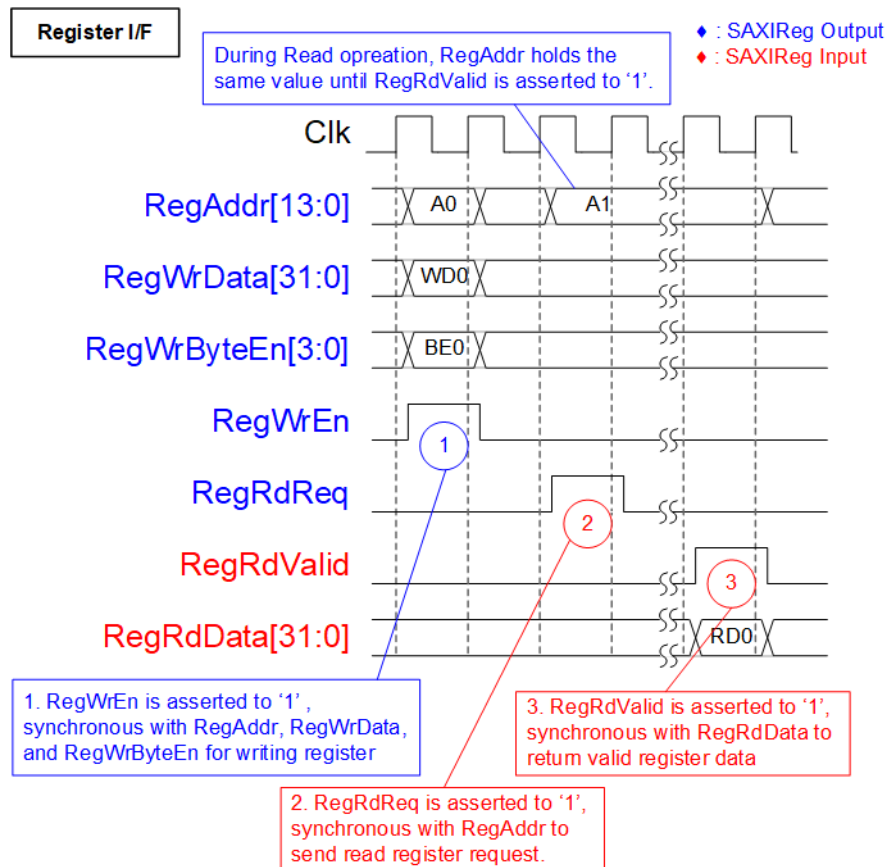


Figure 2-13 Register interface timing diagram

- 1) To write register, the timing diagram is similar to single-port RAM interface. RegWrEn is asserted to '1' with the valid signal of RegAddr (Register address in 32-bit unit), RegWrData (write data of the register), and RegWrByteEn (the write byte enable). Byte enable has four bits to be 4-byte data enable. 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 '1' with the valid value of RegAddr. 32-bit data must be returned after receiving the read request. The slave must monitor RegRdReq signal to start the read transaction. During read operation, the address value (RegAddr) does not change the value until RegRdValid is asserted to '1'. Therefore, the address can be used for selecting the returned data by using multiple multiplexers.
- 3) The read data is returned on RegRdData bus by the slave with asserting RegRdValid to '1'. After that, SAXIReg forwards the read value to LAXir* interface.

2.2.2 UserReg

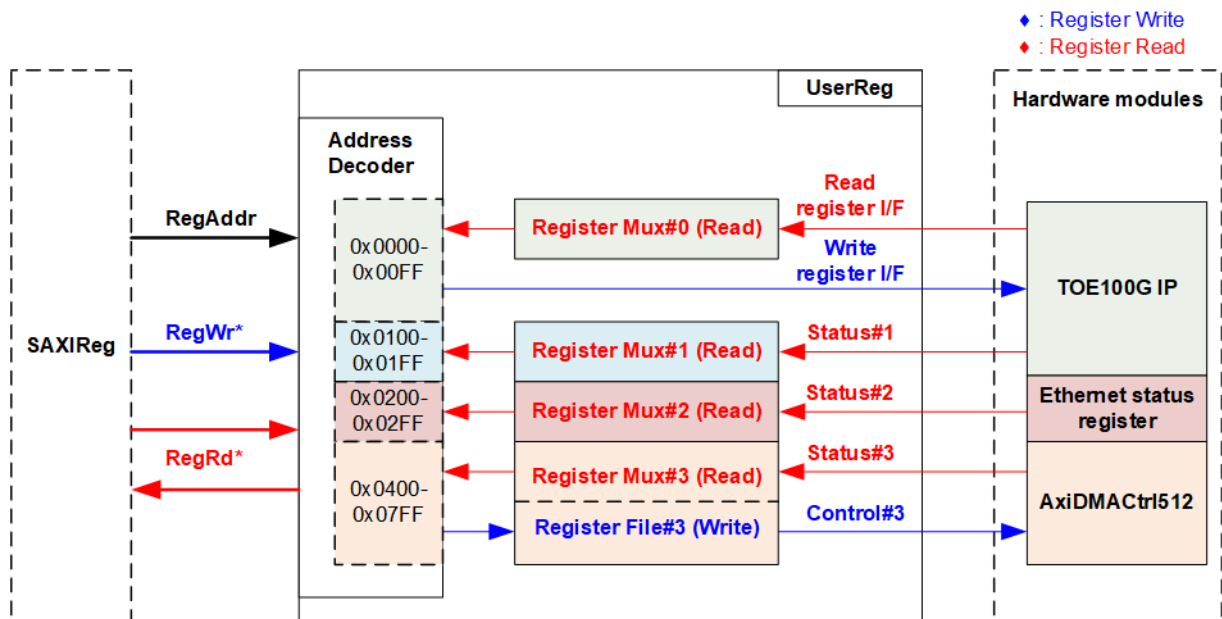


Figure 2-14 UserReg block diagram

UserReg consists of many registers for interfacing with the hardware submodules, i.e., TOE100G-IP, Ethernet system, and AxiDMACtrl512. The address for write or read access is decoded by Address decoder to select the active register. There are four addressing areas, as shown in Figure 2-14.

- 1) 0x0000 – 0x00FF: TOE100G-IP register interface area
- 2) 0x0100 – 0x01FF: TOE100G-IP status area
- 3) 0x0200 – 0x02FF: Ethernet system status area
- 4) 0x0400 – 0x07FF: AxiDMACtrl512 control and status area

Address decoder decodes the upper bits of RegAddr for selecting the active address area while the lower bits is applied to select the active register in each area. The register file inside UserReg is 32-bit data size and write byte enable (RegWrByteEn) is not used, so the CPU must use 32-bit pointer for writing these registers. There are many status registers in UserReg, so multi-level multiplexers are applied to return the read value. In this design, the latency time of read data is equal to four clock cycles, so RegRdValid is created by RegRdReq with asserting four D Flip-flops. More details of the address mapping within UserReg module are shown in Table 2-1.

Table 2-1 Register map Definition

Address Wr/Rd	Register Name (Label in the TOE100DMATest.cpp) Description
BA+0x0000 – BA+0x00FF: TOE100GIP register interface (Write/Read access)	
BA+0x0000	DG_TOEIP_RST_INTREG_OFFSET Mapped to RST register within TOE100G-IP.
BA+0x0004	DG_TOEIP_CMD_INTREG_OFFSET Mapped to CMD register within TOE100G-IP.
BA+0x0008	DG_TOEIP_SML_INTREG_OFFSET Mapped to SML register within TOE100G-IP.
BA+0x000C	DG_TOEIP_SMH_INTREG_OFFSET Mapped to SMH register within TOE100G-IP.
BA+0x0010	DG_TOEIP_DIP_INTREG_OFFSET Mapped to DIP register within TOE100G-IP.
BA+0x0014	DG_TOEIP_SIP_INTREG_OFFSET Mapped to SIP register within TOE100G-IP.
BA+0x0018	DG_TOEIP_DPN_INTREG_OFFSET Mapped to DPN register within TOE100G-IP.
BA+0x001C	DG_TOEIP_SPN_INTREG_OFFSET Mapped to SPN register within TOE100G-IP.
BA+0x0020	DG_TOEIP_TDL_INTREG_OFFSET Mapped to TDL register within TOE100G-IP.
BA+0x0024	DG_TOEIP_TMO_INTREG_OFFSET Mapped to TMO register within TOE100G-IP.
BA+0x0028	DG_TOEIP_PKL_INTREG_OFFSET Mapped to PKL register within TOE100G-IP.
BA+0x002C	DG_TOEIP_PSH_INTREG_OFFSET Mapped to PSH register within TOE100G-IP.
BA+0x0030	DG_TOEIP_WIN_INTREG_OFFSET Mapped to WIN register within TOE100G-IP.
BA+0x0034	DG_TOEIP_ETL_INTREG_OFFSET Mapped to ETL register within TOE100G-IP.
BA+0x0038	DG_TOEIP_SRV_INTREG_OFFSET Mapped to SRV register within TOE100G-IP.
BA+0x003C	DG_TOEIP_VER_INTREG_OFFSET Mapped to VER register within TOE100G-IP.
BA+0x0040	DG_TOEIP_DML_INTREG_OFFSET Mapped to DML register within TOE100G-IP.
BA+0x0044	DG_TOEIP_DMH_INTREG_OFFSET Mapped to DMH register within TOE100G-IP.
BA+0x0100 – BA+0x01FF: TOE100GIP status (Write/Read access)	
BA+0x0100	TOE100G-IP status DG_TOEIP_USERSTS_INTREG_OFFSET Wr – [8]: Asserted to '1' to clear this bit which shows the latched value of TimerInt. Rd – [0]: Mapped ConnOn from TOE100G-IP. Rd – [8]: Latched value of TimerInt output from IP ('0': Normal, '1': TimerInt='1' is detected).
BA+0x0110	Connection interrupt DG_TOEIP_USERINT_INTREG_OFFSET Wr – [0]: Set '1' to clear the connection interrupt. Rd – [0]: Interrupt from ConnOn edge detection. ('1': Detect edge of ConnOn signal from TOE100G-IP, '0': ConnOn does not change the value.) <i>Note: ConnOn value can be read from DG_TOEIP_CONNON_INTREG_OFFSET.</i>

Address Wr/Rd	Register Name (Label in the TOE100DMATest.cpp) Description
BA+0x0200 – BA+0x02FF: Ethernet status (Read access only)	
BA+0x0200	Ethernet linkup status DG_EMAC_USERSTS_INTREG_OFFSET Rd – [0]: Ethernet linkup status from 100G Ethernet MAC ('0'- Not linkup, '1'- Linkup).
BA+0x0204	IP Version of DG EMAC-IP DG_EMAC_USERVER_INTREG_OFFSET Rd – [31:0]: Mapped to IPVersion of DG-EMAC. Not used when DG-EMAC is not implemented.
BA+0x0400 – BA+0x07FF: AxiDMACtrl512 control and status (Write/Read access) <i>Note: BA+0x0600 – BA+0x06FF: Tx buffer parameters [Host -> Card] BA+0x0700 – BA+0x07FF: Rx buffer parameters [Card -> Host]</i>	
BA+0x0400	AxiDMACtrl512 reset DG_DMA_RESET_OFFSET Wr/Rd – [0]: Reset signal to AxiDMACtrl512 module ('1'-Reset, '0'-Clear).
BA+0x0404	AxiDMACtrl512 command DG_DMA_COMMAND_OFFSET Wr – [0]: Start Tx transfer. Asserted to '1' to start Tx transfer on AxiDMACtrl512. This flag is auto-cleared. Wr – [1]: Start Rx transfer. Asserted to '1' to start Rx transfer on AxiDMACtrl512. This flag is auto-cleared.
BA+0x0408	AxiDMACtrl512 status DG_DMA_STATUS_OFFSET Rd – [0]: Tx transfer busy flag. Asserted to '1' when AxiDMACtrl512 is operating Tx transfer. Rd – [1]: Rx transfer busy flag. Asserted to '1' when AxiDMACtrl512 is operating Rx transfer. Rd – [9:8]: The active area of Tx buffer that is in 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 in operating. (00b-Rx buffer#0, 01b-Rx buffer#1, 10b-Rx buffer#2, 11b-Rx buffer#3)
BA+0x0410	Total transmit length of AxiDMACtrl512 DG_DMA_TOTAL_TRANSMIT_LENGTH_OFFSET Wr – [31:0]: Total amount of data for Tx transfer in 512-bit unit. Valid range is 1-0xFFFFFFFF. Rd – [31:0]: Current amount of data that is completely transmitted in 512-bit unit. Valid while operating.
BA+0x0414	Total receive length of AxiDMACtrl512 DG_DMA_TOTAL_RECEIVE_LENGTH_OFFSET Wr – [31:0]: Total amount of data for Rx transfer in 512-bit unit. Valid range is 1-0xFFFFFFFF. Rd – [31:0]: Current amount of data that is completely received in 512-bit unit. Valid while operating.
BA+0x0418	Tx buffer status of AxiDMACtrl512 DG_DMA_TXBUFFER_VALID_OFFSET Wr/Rd – [3:0]: Each bit is mapped to show the status of each area for Tx buffer. Bit[0], [1], [2], and [3] show the status of Tx buffer#0, #1, #2, and #3, respectively. Wr: Asserted to '1' when the data in Tx buffer#i is ready for Tx transfer. Rd: '0'-No data stored in Tx buffer#1, '1'-Has data stored in Tx buffer#i. In Tx transfer, this flag is asserted by CPU when the data is completely prepared. It is de-asserted by the hardware kernel when all data is completely read.
BA+0x041C	Rx buffer valid of AxiDMACtrl512 DG_DMA_RXBUFFER_VALID_OFFSET Wr/Rd – [3:0]: Each bit is mapped to show the status of each area for Rx buffer. Bit[0], [1], [2], and [3] show the status of Rx buffer#0, #1, #2, and #3, respectively. Wr: Asserted to '1' to clear this bit which shows data ready status of Rx buffer#i. Rd: '0'-No data stored in Rx buffer#1, '1'-Has data stored in Rx buffer#i. In Rx transfer, this flag is asserted by the hardware kernel when the data is completely prepared. It is de-asserted by CPU when all data is completely read.
BA+0x0480	Buffer size of AxiDMACtrl512 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 512 bits, so bit[5:0] of this register must be equal to 000000b. <i>Note: The hardware kernel loads this register when the reset flag (DG_DMA_RESET_OFFSET) is asserted.</i>

Address Wr/Rd	Register Name (Label in the TOE100DMATest.cpp) Description
BA+0x0400 – BA+0x07FF: AxiDMACtrl512 control and status (Write/Read access) Note: BA+0x0600 – BA+0x06FF: Tx buffer parameters [Host -> FPGA] BA+0x0700 – BA+0x07FF: Rx buffer parameters [FPGA -> Host]	
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	32-bit lower base address of Tx buffer#0 for AxiDMACtrl512 DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(0) Wr/Rd - [31:0]: Mapped to the 32-bit lower address of Tx buffer#0 in the host memory. Loaded while AxiDMACtrl512 reset is active.
BA+0x0604	32-bit higher base address of Tx buffer#0 for AxiDMACtrl512 DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(0) Wr/Rd - [31:0]: Mapped to the 32-bit higher address of Tx buffer#0 in the host memory. Loaded while AxiDMACtrl512 reset is active.
BA+0x0610 - BA+0x061F	Tx buffer#1 parameters for AxiDMACtrl512 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 AxiDMACtrl512 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 AxiDMACtrl512 0x0630: DG_DMA_TXBUFFER_LOW_ADDRESS_OFFSET(3) 0x0634: DG_DMA_TXBUFFER_HIGH_ADDRESS_OFFSET(3)
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	32-bit lower base address of Rx buffer#0 for AxiDMACtrl512 DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(0) Wr/Rd - [31:0]: Mapped to the 32-bit lower address of Rx buffer#0 in the host memory Loaded while AxiDMACtrl512 is in reset phase.
BA+0x0704	32-bit higher base address of Rx buffer#0 for AxiDMACtrl512 DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(0) Wr/Rd - [31:0]: Mapped to the 32-bit higher address of Rx buffer#0 in the host memory. Loaded while AxiDMACtrl512 is in reset phase.
BA+0x0710 - BA+0x071F	Rx buffer#1 parameters for AxiDMACtrl512 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 AxiDMACtrl512 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 AxiDMACtrl512 0x0730: DG_DMA_RXBUFFER_LOW_ADDRESS_OFFSET(3) 0x0734: DG_DMA_RXBUFFER_HIGH_ADDRESS_OFFSET(3)

3 The host software

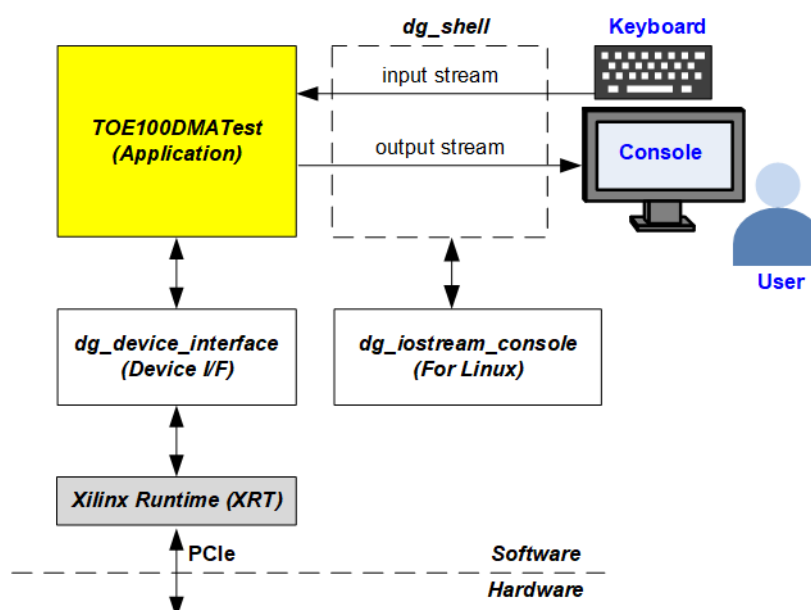


Figure 3-1 The software architecture in TOE100G-IP on Alveo card demo

The host software for this demo consists of two software categories – the application and the framework. “TOE100DMATest” is the main application of this demo. While the framework has three source codes. First is “dg_shell” which handles the user input (keyboard) and the output console (monitor). The input stream and the output stream on the Linux OS has its own control sequence, so the second framework – “dg_iostream_console” is designed by using specific command for Linux OS to handle the stream. Last is “dg_device_interface” which is applied to control the hardware interface on Alveo card by using Xilinx runtime (XRT). It includes the functions to write/read hardware registers and handle the process for memory allocation.

Xilinx Runtime Library (XRT) is the software interface for communicating between the application and the hardware kernels. When the hardware kernels are implemented on Alveo card, the interface is based on PCIe. More details about the Xilinx Runtime Library can be found from the following link.

<https://www.xilinx.com/products/design-tools/vitis/xrt.html>

More details of the software on the demo are describes as follows.

3.1 Framework

There are two software frameworks that are designed for this demo, i.e., the device interface and the shell. The device interface framework makes a simple function of utilizing the Xilinx Runtime (XRT) for interfacing with the hardware kernels. While the shell framework handles the input and output of the console (Linux terminal) for user interface.

3.1.1 Device interface

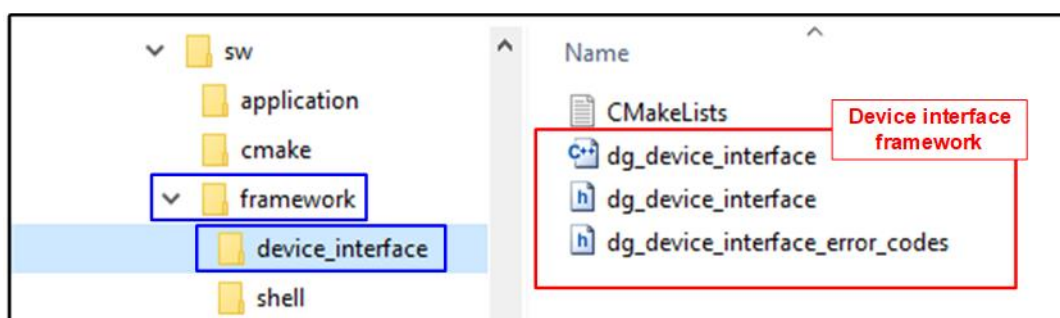


Figure 3-2 Device interface framework

The device interface is used by the application for communicating with the hardware kernels through Xilinx Runtime Library (XRT). The application uses it to create a connection, to allocate the buffer, and to write/read the hardware registers. As shown in Figure 3-2, there are three source codes inside device_interface directories.

- “dg_device_interface_error_codes.h”: Define the returned value of the function for the device interface which may be “OK” status or error codes. The returned value of some functions in this class is referred to these definitions. The values are listed as follows
 - 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”: Declare a class and functions which are defined in C++ source file (dg_device_interface.cpp). The header file determines which function can be or cannot be called from other class. Also, it declares variables in the class with the initial value if it is specified.
- “dg_device_interface.cpp”: Design the general function for connecting the device, accessing the hardware register, and allocating/de-allocating the host memory. The function lists of the device interface framework are described as follows.

Note: The string of compute unit name (cu_name) is defined as the constant in the software source code – “TOE100DMATest:TOE100DMATest”. 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 (TOE100DMATest:TOE100DMATest). 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

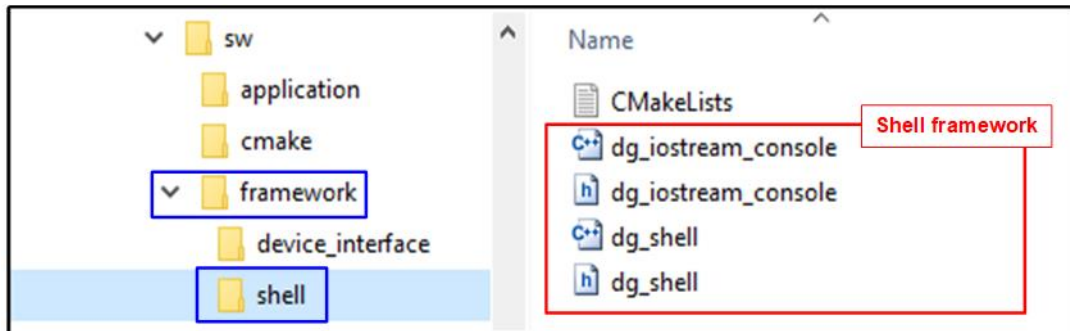


Figure 3-3 Shell framework

The shell framework (dg_shell) handles the input and output stream on the Linux terminal (Console). It retrieves keyboard input, manages the input string, parses the input data type, and prints a string out to console. The shell framework uses an I/O stream console library (dg_iostream_console) to work with the Linux terminal, i.e., changing the terminal environment, getting the user keyboard input, and pushing the printed string output to terminal.

As shown in Figure 3-3, there are two source codes for handling I/O stream console.

- “dg_iostream_console.h”: Declare a class and functions which are defined in C++ source file (dg_iostream_console.cpp). The header file determines which function can be or cannot be called from other class. Also, it declares variables in the class with the initial value if it is specified.
- “dg_iostream_console.cpp”: Design the general function to manage the input and output stream on the Linux terminal environment such as writing a string on console, changing the terminal setting for utilizing by the shell, reverting the terminal setting to the original one, and getting the input character from the user through terminal. The function lists of the I/O stream classes are described as follows.

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). It contains the keyboard input type for processing in the shell framework which are NORMAL, BACKSPACE, LEFTARROW, RIGHTARROW, DELETE, TAB, EOL, and CONTROL.

OutputStreamConsole 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 setting to non-echo mode and to process an input from the terminal without endline character. The original setting is stored to a local variable for retrieving later.

void RestoreSetting(void)	
Parameters	None
Return value	None
Description	Restore the Linux terminal setting to the original setting 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 using this function, 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 has 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 has two source codes, described as follows.

- “dg_shell.h”: Similar to the “dg_iostream_console.h” header file, it declares a class, functions and variables which are defined in C++ source file (dg_shell.cpp).
- “dg_shell.cpp”: Design the general function to simplify the input and output console management function and to provide a utility function such as parsing a string to an unsigned integer. The function lists of the shell framework are described as follows.

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	Clear the input buffer and the input stream by using “ClearInputBuffer” and “FlushInputStream”. After that, receive the input from the terminal (use “getChar”) and then process the input by using “ProcessInputChar”. The input is read and processed until end-of-line is detected. 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 is applied to map the function of “dg_iostream_console” for the application using.

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 by using "GetInputCharLength". Return "True" if the length is greater than zero. 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	This function is called to receive the input string with the argument and then calculate the length for writing to the output stream. It is almost similar to standard "printf" function, but displaying to the console through the output stream.

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” 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 “keyPressEvent” (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 show the end of line flag to be "true". When "GetInputLine" 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

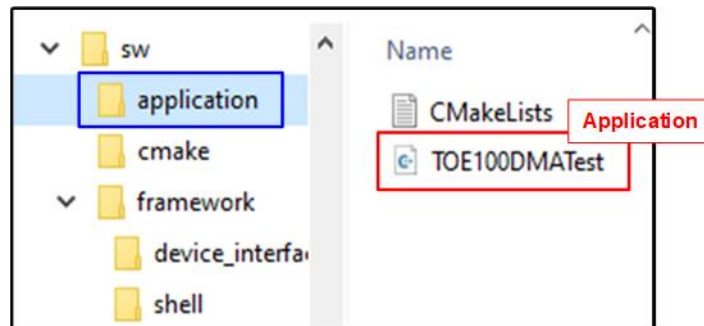


Figure 3-4 Application layer

The source code of the application is “TOE100DMATest.cpp” file, as shown in Figure 3-4. The main function is operated by following steps.

- 1) Start a signal handler to detect “CTRL+C” input from the user. If the key is found, the termination process is run as below.
 - i) Reset the hardware kernel (TOE100GDMATest).
 - ii) De-allocate the memory via XRT.
 - iii) Close the device interface.
 - iv) Restore the terminal setting to the original setting.
- 2) Connect the hardware platform through device interface framework. Setup the system to interface with the hardware kernel and the Linux terminal.
- 3) Examine that TOE100G-IP is available in the hardware kernel and print the IP information.
- 4) Prepare the memory for the DMA feature by allocating 1 GB memory in the host system through the device interface.
- 5) Setup input/output stream and the Linux terminal for the application using the shell object.
- 6) Start initializing TOE100G-IP in the hardware platform as below.
 - i) Wait until Ethernet link is established, read by DG_EMAC_STS_INTREG_OFFSET[0] register.
 - ii) Display welcome message and then wait for the input from user to select the initialization mode of TOE100G-IP. Three modes can be selected – Client, Server, or Fixed-MAC.
 - If the target device on another side of 100G Ethernet network is the PC, it is recommended to initialize TOE100G-IP to Client mode. TOE100G-IP creates ARP request packet and then the target device returns ARP replay packet.
 - If the target device on another side of 100G Ethernet network is the system that integrates TOE100G-IP, the mode on two TOE100G-IPs can be three formats, i.e., Server – Client, Client – Fixed MAC, or Fixed MAC – Fixed MAC.
 - iii) After user sets the initialization mode, the default parameters of that initialization mode are shown on the console, as shown in Figure 3-5.

```

+++ TOE100GIP with DMA Demo [IPVer = 1.1] +++

Input mode : [0] Client [1] Server [2] Fixed MAC => 0

+++ Current Network Parameter +++
Window Update Gap = 16
Reverse Packet    = ENABLE
Mode              = CLIENT
FPGA MAC address  = 0x000102030405
FPGA IP           = 192.168.100.42
FPGA port number  = 60000
Target IP         = 192.168.100.25
Target port number = 60001
Press 'x' to skip parameter setting: x
IP initialization complete

```

Annotations in the image:

- iii: User input mode (points to '0')
- iii: Display default network parameter (points to the network parameter list)
- iv: Input network parameter (use default) (points to 'x')
- v: IP initialization complete (points to the completion message)

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

- iv) User enters 'x' to start initialization process on TOE100G-IP by using the default parameters or enters other keys to change some parameters (more details are described in topic 3.2.2 Reset menu).
- v) Wait until TOE100G-IP finishes the IP initialization process, checked by `DG_TOEIP_CMD_INTREG[0]='0'`.
- 7) Start initializing DMA environment for both hardware platform and the software application, described as below.
 - i) Assign variable to point to the allocated memory from device interface. There are two pointer types. First is the host virtual memory address pointer that is used in the application and another is the hardware memory address that is used by the hardware. Both pointer types are pointed to the same memory location.
 - ii) Send a soft reset to the DMA logic in the hardware platform (`DG_DMA_RESET_OFFSET[0]='1'`).
 - iii) Set the buffer size to the hardware register (`DG_DMA_BUFFER_SIZE_OFFSET`) and then set the 64-bit hardware memory address of the transmit and receive buffer to the hardware registers (`DG_DMA_TX/RXBUFFER_LOW/HIGH_ADDRESS_OFFSET`). This reference design uses four memory areas, so the hardware registers of four areas are set.
 - iv) De-assert the soft reset to DMA logic (`DG_DMA_RESET_OFFSET[0]='0'`).
- 8) After the hardware completes the initialization process, the main menu of the application is displayed on the console, as shown in Figure 3-6. There are five test operations for user selection. More details of each menu are described as follows.

```

Press 'x' to skip parameter setting: x
IP initialization complete

--- TOE100G-IP menu ---
[0] : Display TCPIP parameters
[1] : Reset TCPIP parameters
[2] : Send Data Test (TOEIP -> Target)
[3] : Receive Data Test (Target -> TOEIP)
[4] : Full duplex Test (TOEIP <-> Target)
>>>

```

Figure 3-6 Main menu of the software application

3.2.1 Display parameters

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

- 1) Read all network parameters from each variable in the software application.
- 2) Print out each variable.

3.2.2 Reset IP

This menu is applied to change network parameters of TOE100G-IP such as IP address and source port number. The software application asserts the TOE100G-IP reset before updating the parameters to TOE100G-IP register. After that, the reset is de-asserted to start IP initialization by using the updated parameters. The software application monitors busy flag to wait until the initialization is completed. The sequence to reset IP is as follows.

- 1) Display current parameter value on the console.
- 2) Ask user to skip (use current parameters) or set new parameter values.
 - i) Press 'x' on keyboard to skip. The current parameters are used and continue to step 6.
 - ii) Press other keys to start setting parameters (continue to step 3).
- 3) Receive initialization mode from user.
 - i) If input mode is invalid or the input mode does not change, mode value will not change and continue to step 4.
 - ii) If input mode is valid and changes from previous value, the current parameter set of new mode is displayed on the console. Next, user inputs 'x' to use the current parameters (continue to step 6) or inputs other keys to adjust some parameters (continue to step 4).
- 4) Receive all the remaining parameters from user, i.e., FPGA MAC address and FPGA IP address. If an input value is invalid, the parameter is not changed.
- 5) Force reset to IP by setting DG_TOEIP_RST_INTREG_OFFSET[0]='1'.
- 6) Set all parameters to TOE100G-IP register such as DG_TOEIP_SML_INTREG_OFFSET and DG_TOEIP_DIP_INTREG_OFFSET.
- 7) De-assert IP reset by setting DG_TOEIP_RST_INTREG_OFFSET[0]='0'. After that, TOE100G-IP starts the initialization process.
- 8) Monitor IP busy flag (DG_TOEIP_CMD_INTREG_OFFSET[0]) until the initialization process is completed (busy flag is de-asserted to '0').

3.2.3 Send data test

Four user inputs are received to set total transmit length, packet size, pattern data generation mode (enable or disable), and connection mode (active open for client operation or passive open for server operation). Before running the test, the software application creates a child thread to write the 32-bit incremental data or dummy data to the transmit buffer (Tx buffer). After that, the main thread sets control flag to the hardware register for reading the data from the Tx buffer and then transferring to the TOE100G-IP for creating the data packet to the target. The operation is finished when total data are transferred from the system to the target completely.

To handle the flow control of the Tx buffer, three counters are applied. First is the write counter of Tx buffer which is increased by the child thread after finishing writing data of each Tx buffer area. Second is the request counter that is increased by the main thread after sending the request to the hardware to start reading the new data from each Tx buffer area. Third is the read counter of 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. The sequence of Send data test menu is as follows.

- 1) Receive transfer size, packet size, data generation mode, and connection mode from user and verify if all inputs are valid.
- 2) Read current parameters and then display the recommended parameters of “tcpdatatest” (Test application on the Target PC) when the target device is another PC.
- 3) Open connection following connection mode setting.
 - i) For active open, the software application sets DG_TOEIP_CMD_INTREG_OFFSET=2 (Open port) and waits until ConnOn status is changed by checking the connection interrupt status (DG_TOEIP_USERINT_INTREG_OFFSET[0]='1'). Error message is displayed if TOE100G-IP finishes the operation without interrupt asserting.
 - ii) For passive open, the software application waits until connection is opened by the Target device. Connection interrupt status (DG_TOEIP_USERINT_INTREG_OFFSET[0]) is monitored until it is equal to '1'.
- 4) Prepare the parameters and the software application for Transmit DMA feature.
 - i) Calculate the amount of data for the last Tx buffer area and the total count of Tx buffer areas for storing all data.
 - ii) Create a child thread (gen_txbuf_data) for writing the data into the Tx buffer until all data is filled completely.
- 5) Setup the hardware for DMA function.
 - i) Set the total transmit length to DG_DMA_TOTAL_TRANSMIT_LENGTH_OFFSET.
 - ii) Start Tx DMA engine hardware by setting DG_DMA_COMMAND_OFFSET[0]='1'.
 - iii) Set packet size to TOE100G-IP register (DG_TOEIP_PKL_INTREG_OFFSET).

- 6) Control the data transmission of Tx buffer with TOE100G-IP and Tx DMA engine by running the following steps as forever loop until all data is transmitted completely or some errors are found.
 - i) Check if all data are completely transferred to TOE100G-IP. If not, continue to the next step to prepare the next data transmission. Otherwise, skip to step 7).
 - ii) Check if TOE100G-IP finishes the operation of the previous Send command by reading busy flag (DG_TOEIP_CMD_INTREG_OFFSET[0]='0') and remaining transfer length (not equal to 0). If not, continue to the next step. Otherwise, prepare the next Send command parameters for TOE100G-IP. The transfer size of TOE100G-IP (DG_TOEIP_TDL_INTREG_OFFSET) is set to 4 GB, except the last loop that can be less than 4 GB. After that, set the Send command to TOE100G-IP (DG_TOEIP_CMD_INTREG_OFFSET[0]=0).
 - iii) Check if there is new data filled by the child thread (request counter < write counter). If not, continue to the next step. Otherwise, set the valid flag of the new Tx buffer area to the hardware (DG_DMA_TXBUFFER_VALID_OFFSET[i]='1' where 'i' is an index of the new buffer area) and then increase the request counter.
 - iv) Check if there is new clear flag of Tx buffer returned by the hardware after finishing reading the data from Tx buffer (DG_DMA_TXBUFFER_VALID_OFFSET). If not, continue to the next step. Otherwise, increase the read counter.
 - v) Display the results on the console every second and return to step i).
- 7) Wait until the TOE100G-IP completes its operation by monitoring the busy flag (DG_TOEIP_CMD_INTREG_OFFSET[0]='0').
- 8) Set close connection command to TOE100G-IP register (DG_TOEIP_CMD_INTREG_OFFSET=3) and wait until the operation is successful by reading the connection interrupt status (DG_TOEIP_USERINT_INTREG_OFFSET[0]='1'). Error message is displayed if TOE100G-IP finishes the operation without interrupt asserting.
- 9) Wait until the child thread finishes the operation.
- 10) Calculate performance and show test result on the console.

3.2.4 Receive data test

User sets total amount of received data, data verification mode (enable or disable), and connection mode (active open for client operation or passive open for server operation). Before running the test, the software application creates a child thread to read data from the receive buffer (Rx memory) which is written by the hardware. If the data verification mode is enabled, the received data are verified by 32-bit incremental data. The operation is finished when total data are received and the connection is closed by the target.

Similar to Send data test, three counters are applied to handle the flow control of the Rx buffer. First is the write counter of 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. Second is the read counter of Rx buffer which is increased by the child thread after finishing reading data of each Rx buffer area. Third is the complete counter which is increased by the main thread after setting the clear flag to each Rx buffer area. The sequence of Receive data test menu is as follows.

- 1) Receive total transfer size, data verification mode, and connection mode from user input. Verify that all inputs are valid.
- 2) Display the recommended parameters for running “tcpdatatest” when the target device is another PC, similar to step 2 of Send data test.
- 3) Open connection following connection mode setting, similar to step 3 of Send data test.
- 4) Prepare the parameters and the software application for Receive DMA feature.
 - i) Calculate the amount of data for the last Rx buffer area and the total count of the Rx buffer areas for storing all data.
 - ii) Create a child thread (ver_rxbuf_data) for reading and verifying the data from the Rx buffers until all data is read completely.
- 5) Setup the hardware for DMA function.
 - i) Set the total receive length to DG_DMA_TOTAL_RECEIVE_LENGTH_OFFSET.
 - ii) Start Rx DMA engine hardware by setting DG_DMA_COMMAND_OFFSET[1]='1'.
- 6) Control the data transmission of Rx buffer with TOE100G-IP and Rx DMA engine by running the following steps as forever loop until the connection is terminated or some errors are found.
 - i) Check if the connection is terminated. If not, continue to the next step. Otherwise, wait until the Rx DMA engine completes the operation (DG_DMA_STATUS_OFFSET[1]='0') and there is no more data in the Rx buffer for reading (DG_DMA_RXBUFFER_VALID_OFFSET=0). After that, skip to the step 7).
 - ii) Check if there is the new valid flag of Rx buffer (DG_DMA_RXBUFFER_VALID_OFFSET[i]='1' where 'i' 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, continue to the next step. Otherwise, increase the write counter. 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.
 - iii) Check if the read counter is updated by the child thread (read counter > complete counter). If not, continue to the next step. Otherwise, set clear flag of the new Rx buffer area to the hardware (DG_DMA_RXBUFFER_VALID_OFFSET[i]='1' where 'i' is an index of the new buffer area). After that, increase the complete counter by the main thread.
 - iv) Display the results on the console every second and return to step i).
- 7) Wait until the child thread finishes the operation.
- 8) Display the error messages if the errors have occurred (data verification fail or total received length mismatch).
- 9) Calculate performance and show test result on the console.

3.2.5 Full duplex test

This menu transfers the data between the system and the target device in both directions by using the same port number at the same time. Four inputs are received from user, i.e., total data size for both transfer directions, transmit packet size, data generation/verification mode, and connection mode (active open/close for client operation or passive open/close for server operation). When running the test, the transfer size that is set on the system and the target device must be equal. If the target device is PC that runs “tcp_client_txrx(_40G)” application, the system must set the connection mode to be passive. The test runs in forever loop until the user cancels operation by entering any keys on console at the end of each test round.

Two child threads are created for handling the transmit buffer (Tx buffer) and the receive buffer (Rx buffer) individually. The flow control of each buffer is handled by using three counters. For Tx buffer, it uses Tx write counter, Tx request counter, and Tx read counter. While Rx buffer uses Rx write counter, Rx read counter, and Rx complete counter. The function of the counter is similar to the description in Send data test and Receive data test. The sequence of this test is as follows.

- 1) Receive total data size, packet size, data generation/verification mode, and connection mode from the user and verify that all inputs are valid.
- 2) Display the recommended parameters for running “tcp_client_txrx_40G” when the target device is another PC, similar to step 2 of Send data test.
- 3) Open connection following connection mode setting, similar to step 3 of Send data test.
- 4) Prepare the parameters and the software application for Tx DMA and Rx DMA features.
 - i) Calculate the amount of data for the last Tx/Rx buffer area and total count of the Tx/Rx buffer areas for storing all data.
 - ii) Set packet size to TOE100G-IP register (DG_TOEIP_PKL_INTREG_OFFSET).
 - iii) Create two child threads - gen_txbuf_data and ver_rxbuf_data for writing the data to Tx buffers and reading and verifying the data from Rx buffers, respectively.
- 5) Setup the hardware for DMA function.
 - i) Set the total transmit length and total receive length to the hardware registers (DG_DMA_TOTAL_TRANSMIT/RECEIVE_LENGTH_OFFSET) by the same value.
 - ii) Start Tx and Rx DMA engine hardware by setting DG_DMA_COMMAND_OFFSET=3.
- 6) Control the data transmission of Tx buffer and Rx buffer with TOE100G-IP and DMA engine by running the following steps as forever loop until all data is transferred completely or some errors are found.
 - i) Check the condition to exit the loop which is different for the active mode and passive mode. If the exit condition is met, skip to step 7). Otherwise, continue to the next step to prepare the next data transmission. The exit condition of each mode is as follows.
 - a. For active mode, check read counter of Tx buffer and Tx remaining length to confirm that all Tx data are transferred completely. Also, check that Rx buffer status is in Idle condition and all flags of Rx buffer are cleared.
 - b. For passive mode, check that the connection is terminated. Also, Rx buffer status must be Idle and all flags of Rx buffer are cleared. Error is found if there is remaining data that is not transferred in Tx buffer.

- ii) Check if TOE100G-IP finishes the operation of the previous Send command by reading busy flag (DG_TOEIP_CMD_INTREG_OFFSET[0]='0') and Tx remaining length (not equal to 0). If not, continue to the next step. Otherwise, prepare the next Send command parameters for TOE100G-IP. The transmit size of TOE100G-IP (DG_TOEIP_TDL_INTREG_OFFSET) is set to the maximum value that is less than 4 GB and aligned to the transmit packet size, except the last loop that is set by Tx remaining length. After that, set the Send command to TOE100G-IP (DG_TOEIP_CMD_INTREG_OFFSET[0]=0).
- iii) Check if there is new Tx data filled by the child thread (Tx request counter < Tx write counter). If not, continue to the next step. Otherwise, set the valid flag of the new Tx buffer area to the hardware (DG_DMA_TXBUFFER_VALID_OFFSET[i]='1' where 'i' is an index of the new buffer area) and then increase the request counter.
- iv) Check if there is new clear flag of Tx buffer returned by the hardware after finishing reading the data from Tx buffer (DG_DMA_TXBUFFER_VALID_OFFSET). If not, continue to the next step. Otherwise, increase the Tx read counter.
- v) Check if there is the new valid flag of Rx buffer (DG_DMA_RXBUFFER_VALID_OFFSET[i]='1' where 'i' 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, continue to the next step. Otherwise, increase the write counter. When the child thread detects the updated Rx write counter, it starts reading and verifying the data from Rx buffer (if the verification flag is set). Rx read counter is updated by the thread after finishing the operation.
- vi) Check if Rx read counter is updated by the child thread (Rx read counter > Rx complete counter). If not, continue to the next step. Otherwise, set clear flag of the new Rx buffer area to the hardware (DG_DMA_RXBUFFER_VALID_OFFSET[i]='1' where 'i' is an index of the new buffer area). After that, increase Rx complete counter by the main thread.
- vii) Display the results on the console every second and return to step i).
- 7) Wait until the TOE100G-IP completes its operation by monitoring the busy flag (DG_TOEIP_CMD_INTREG_OFFSET[0]='0').
 - 8) Set close connection command to TOE100G-IP register if the connection mode is active mode, similar to step 8 of Send data test.
- 9) Wait until the child threads (gen_txbuf_data and ver_rxbuf_data) finish the operation.
- 10) Display the error messages if the errors have occurred (data verification fail or total transfer length mismatch).
- 11) Calculate performance and show test result on the console.
- 12) Display the message and wait for 2 seconds to allow the user entering the keys to end the operation. If some keys are detected, exit the test. Otherwise, return to step 3) to repeat the test.

3.2.6 Function list in application

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

Timer Utilization Function

static void TimerStart(void)	
Parameters	None
Return value	None
Description	Start the clock and store the initial time for returning the elapsed time.

static void TimerStop(void)	
Parameters	None
Return value	None
Description	Stop the clock and store the current time to calculate the elapsed time.

template<typename dur_unit> static double TimerElapsed(void)	
Parameters	None
Return value	The elapsed time value
Description	Determine whether the clock is running or not. If the clock is stopped, return the total time usage that is measured from calling TimerStart function to calling TimerStop function. Otherwise, return the elapsed time since the clock has been started. <i>Note: Time unit can be specified when calling the function. For example, uses "timer.elapsed<std::chrono::seconds>()" to return in second unit.</i>

Termination Function

void cleanup(void)	
Parameters	None
Return value	None
Description	This function is called for safety termination of the application. It resets the hardware system by setting DG_DMA_RESET_OFFSET and DG_TOEIP_RST_INTREG_OFFSET to '1'. Next, de-allocate the memory back to host via XRT. After that, close the target device interface. Finally, restore the terminal setting by using the original setting.

void sigintHandler(void)	
Parameters	None
Return value	None
Description	This function is run to terminate the application when user input "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.

void show_cursize(uint64_t tx_len, uint64_t rx_len)	
Parameters	tx_len: Current transfer size of the transmitted data in byte unit rx_len: Current transfer size of the received data in byte unit
Return value	None
Description	Display the current amount of transmitted data and received data in Byte, KByte, or MByte unit.

void show_result(uint64_t tot_tx_len, uint64_t tot_rx_len, double time_val)	
Parameters	tot_tx_len: Total transfer size of the transmitted data in byte unit tot_rx_len: Total transfer size of the received data in byte unit time_val: Total time usage in millisecond unit
Return value	None
Description	Display total amount of transmitted data and received data. Next, Display total time usage in sec unit. Finally, transfer performance is calculated and displayed in MB/s unit.

void show_ipaddr(uint32_t ip_addr)	
Parameters	ip_addr: 32-bit IPv4 address
Return value	None
Description	Display IPv4 address in decimal unit, separated by dot character.

Thread Function

void kill_thread(std::thread &t)	
Parameters	t: Pointer to a thread function that wanted to terminate
Return value	None
Description	Use to terminate the thread function immediately. Generally, this function will be used when an error has occurred only.

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 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 gen_patt: TRUE-fill the pattern data, FALSE-not fill the pattern data
Return value	None
Description	The operation is exited if buf_wrcnt is more than or equal to buf_totalcnt. Before filling the new data, check that the Tx buffer is not full by reading buf_wrcnt and buf_rdcnt. After that, calculate the start address to write the data and total transfer length of this loop. If gen_patt is TRUE, fill the incremental pattern to Tx buffer. Otherwise, skip the step to fill the data to Tx buffer. Finally, increase the buf_wrcnt value.

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 bool ver_en)	
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 ver_en: TRUE-verify data, FALSE-not verify data
Return value	None
Description	The operation is exited if buf_rdcnt is more than or equal to buf_totalcnt. To start the new operation, check that there is the new data stored in Rx buffer (buf_rdcnt is less than buf_wrcnt). After that, calculate the start address to read the data and total transfer length of this loop. If ver_en is TRUE, the read data from Rx buffer is compared to the incremental pattern. Otherwise, skip the step to verify the data in Rx buffer. Finally, increase 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 the data or not by reading rxbuf_ctrl.

Miscellaneous Function

void wait_ethlink(void)	
Parameters	None
Return value	None
Description	Read DG_EMAC_USERSTS_INTREG_OFFSET[0] and wait until ethernet connection is established.

void input_param(void)	
Parameters	None
Return value	None
Description	Receive test parameters from user for test parameters, i.e., Initialization mode, FPGA MAC address, FPGA IP address, the number of targets, ARP/ICMP Enable, Target MAC address (when run in Fixed-MAC mode), Target IP address, FPGA port number, and Target port number. After receiving and verifying all parameters, the current value of all parameters is displayed by calling show_param function.

inline uint32_t read_conon(void)	
Parameters	None
Return value	0: Connection is OFF, 1: Connection is ON
Description	Read value from DG_TOEIP_USERSTS_INTREG_OFFSET register and return only bit0 value to show connection status.

int exec_port(uint32_t port_ctl, uint32_t mode_active)	
Parameters	port_ctl: 1-Open port, 0-Close port mode_active: 1-Active open/close, 0-Passive open/close
Return value	0: The open/close connection is successful -1: Fail to open/close the connection
Description	For active mode, write DG_TOEIP_CMD_INTREG_OFFSET to open or close connection, depending on port_ctl mode. After that, monitor connection status interrupt from bit0 of DG_TOEIP_USERINT_INTREG_OFFSET register until it is asserted. After that, the interrupt flag is cleared.

Test Function

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

void init_param(void)	
Parameters	None
Return value	None
Description	This function is called to set the parameters and reset the IP, following described in topic 3.2.2.


int dma_send_test(volatile uint32_t *mem_ptr)	
Parameters	mem_ptr: Pointer to Tx buffer
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Send data test following description in topic 3.2.3. This function uses gen_txbuf_data as a child thread.

int dma_recv_test(volatile uint32_t *mem_ptr)	
Parameters	mem_ptr: Pointer to Rx buffer
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Receive data test following description in topic 3.2.4. This function uses ver_rxbuf_data as a child thread.

int dma_txrx_test(volatile uint32_t *txmem_ptr, volatile uint32_t *rxmem_ptr)	
Parameters	txmem_ptr: Pointer to Tx buffer rxmem_ptr: Pointer to Rx buffer
Return value	0: The operation is successful -1: Receive invalid input or error is found
Description	Run Full duplex test with DMA following described in topic 3.2.5. This function uses gen_txbuf_data and ver_rxbuf_data as child threads.

4 Test Software on the target

4.1 “tcpdatatest” for half duplex test



```

Command Prompt
D:\Temp>tcpdatatest.exe

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

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

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

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

D:\Temp>_

```

Figure 4-1 “tcpdatatest” application usage

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

- 1) Mode: c – PC runs in Client mode and FPGA runs in Server mode
- 2) Dir: t – transmit mode (PC sends data to FPGA)
r – receive mode (PC receives data from FPGA)
- 3) ServerIP: IP address of FPGA when PC runs in Client mode (default is 192.168.7.42)
- 4) ServerPort: Port number of FPGA when PC runs in Client mode (default is 4000)
- 5) ByteLen: Total transfer size in byte unit. This input is used in transmit mode only and ignored in receive mode. In receive mode, the application is closed when the connection is terminated. In transmit mode, ByteLen must be equal to the total transfer size, set in receive data test menu of FPGA.
- 6) Pattern:
 - 0 – Generate dummy data in transmit mode or disable data verification in receive mode.
 - 1 – Generate incremental data in transmit mode or enable data verification in receive mode.

Note: Window Scale is the optional parameter which is not used in the demo. Also, this parameter is not available in Linux application.

Transmit data mode

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

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

Receive data mode

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

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

4.2 “tcp_client_txrx(_40G)” for full duplex test



```

Command Prompt
D:\Temp>tcp_client_txrx_40G.exe
*****
TCP Tx Rx Version 1.1
*****
tcp_client_txrx_40G.exe [ServerIP] [ServerPort] [ByteLen] [Verification]

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

[Example] tcp_client_txrx_40G.exe 192.168.40.42 60000 137438953440 0

D:\Temp>_

```

Figure 4-2 “tcp_client_txrx_40G” application usage

“tcp_client_txrx_40G” (for Windows OS) or “tcp_client_txrx” (for Linux OS) application is designed to run on PC for sending and receiving TCP data through Ethernet at the same time by using the same port number. The application is run in Client mode, so user needs to input the Server parameters (the network parameters of TOE100G-IP). As shown in Figure 4-2, there are four parameters to run the application, described as follow.

- 1) ServerIP : IP address of FPGA
- 2) ServerPort : Port number of FPGA
- 3) ByteLen : Total transfer size in byte unit. This is total amount of transmitted data and received data. This value must be equal to the transfer size set on FPGA for running full-duplex test.
- 4) Verification:
 - 0 – Generate dummy data for sending function and disable data verification for receiving function. This mode is used to check the best performance of full-duplex transfer.
 - 1 – Generate incremental data for sending function and enable data verification for receiving function.

The sequence of test application is as follows.

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

5 Revision History

Revision	Date	Description
1.0	21-Sep-22	Initial version release